MPI: A Message-Passing Interface Standard

Version 4.0

(Draft)

Unofficial, for comment only

Message Passing Interface Forum

October 30, 2020

1	This document describes the 2019 Draft Specification of the Message-Passing Interface
2	(MPI) standard, intended for comment. It is not an official version of the standard. The
3	MPI standard includes point-to-point message-passing, collective communications, group
4	and communicator concepts, process topologies, environmental management, process cre-
5	ation and management, one-sided communications, extended collective operations, external
6	interfaces, I/O, some miscellaneous topics, and a profiling interface. Language bindings for
7	C and Fortran are defined.
8	Historically, the evolution of the standards is from MPI-1.0 (May 5, 1994) to MPI-1.1 $$
9	(June 12, 1995) to MPI-1.2 (July 18, 1997), with several clarifications and additions and
10	published as part of the MPI-2 document, to MPI-2.0 (July 18, 1997), with new functionality,
11	to MPI-1.3 (May 30, 2008), combining for historical reasons the documents 1.1 and 1.2
12	and some errata documents to one combined document, and to MPI-2.1 (June 23, 2008),
13	combining the previous documents. Version $MPI-2.2$ (September 4, 2009) added additional
14	clarifications and seven new routines. Version $MPI-3.0$ (September 21, 2012) is an extension
15	of MPI-2.2. Version MPI-3.1 (June 4, 2015) adds clarifications and minor extensions to
16	MPI-3.0.
17	
18 19	Comments. Please send comments on MPI to the MPI Forum as follows:
20	1. Subscribe to https://lists.mpi-forum.org/mailman/listinfo/mpi-comments
20	1. Subbilide to hotpot,//11000.mp1 forum.org/mathman/11001110/mp1 commonob
22	2. Send your comment to: mpi-comments@lists.mpi-forum.org, together with the version
23	of the MPI standard and the page and line numbers on which you are commenting.
24	Your comment will be forwarded to MPI Forum committee members for consideration.
25	Messages sent from an unsubscribed e-mail address will not be considered.
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38 39	
40	
41	
42	
43	
44	
45	© 1993, 1994, 1995, 1996, 1997, 2008, 2009, 2012, 2015, 2019 University of Tennessee,
46	Knoxville, Tennessee. Permission to copy without fee all or part of this material is granted,
47	provided the University of Tennessee copyright notice and the title of this document appear,
48	and notice is given that copying is by permission of the University of Tennessee.

Unofficial Draft for Comment Only

2019 Draft Specification, November, 2019. This document contains a draft of the MPI specification as of the date of publication. It has not been adopted as an official MPI specification, and is provided for comment only. This document includes a number of new features that will be present in the final MPI-4.0 document. The largest changes are the addition of persistent collectives, application info assertions, and improvements to the definitions of error handling. In addition, there are a number of smaller improvements and corrections.

Version 3.1: June 4, 2015. This document contains mostly corrections and clarifications to the MPI-3.0 document. The largest change is a correction to the Fortran bindings introduced in MPI-3.0. Additionally, new functions added include routines to manipulate MPI_Aint values in a portable manner, nonblocking collective I/O routines, and routines to get the index value by name for MPI_T performance and control variables.

Version 3.0: September 21, 2012. Coincident with the development of MPI-2.2, the MPI Forum began discussions of a major extension to MPI. This document contains the MPI-3 Standard. This draft version of the MPI-3 standard contains significant extensions to MPI functionality, including nonblocking collectives, new one-sided communication operations, and Fortran 2008 bindings. Unlike MPI-2.2, this standard is considered a major update to the MPI standard. As with previous versions, new features have been adopted only when there were compelling needs for the users. Some features, however, may have more than a minor impact on existing MPI implementations.

Version 2.2: September 4, 2009. This document contains mostly corrections and clarifications to the MPI-2.1 document. A few extensions have been added; however all correct MPI-2.1 programs are correct MPI-2.2 programs. New features were adopted only when there were compelling needs for users, open source implementations, and minor impact on existing MPI implementations.

Version 2.1: June 23, 2008. This document combines the previous documents MPI-1.3 (May 30, 2008) and MPI-2.0 (July 18, 1997). Certain parts of MPI-2.0, such as some sections of Chapter 4, Miscellany, and Chapter 7, Extended Collective Operations, have been merged into the Chapters of MPI-1.3. Additional errata and clarifications collected by the MPI Forum are also included in this document.

Version 1.3: May 30, 2008. This document combines the previous documents MPI-1.1 (June 12, 1995) and the MPI-1.2 Chapter in MPI-2 (July 18, 1997). Additional errata collected by the MPI Forum referring to MPI-1.1 and MPI-1.2 are also included in this document.

Version 2.0: July 18, 1997. Beginning after the release of MPI-1.1, the MPI Forum began meeting to consider corrections and extensions. MPI-2 has been focused on process creation and management, one-sided communications, extended collective communications, external interfaces and parallel I/O. A miscellany chapter discusses items that do not fit elsewhere, in particular language interoperability.

 $\mathbf{5}$

 31

Version 1.2: July 18, 1997. The MPI-2 Forum introduced MPI-1.2 as Chapter 3 in the
 standard "MPI-2: Extensions to the Message-Passing Interface", July 18, 1997. This section
 contains clarifications and minor corrections to Version 1.1 of the MPI Standard. The only
 new function in MPI-1.2 is one for identifying to which version of the MPI Standard the
 implementation conforms. There are small differences between MPI-1 and MPI-1.1. There
 are very few differences between MPI-1.1 and MPI-1.2, but large differences between MPI-1.2
 and MPI-2.

Version 1.1: June, 1995. Beginning in March, 1995, the Message-Passing Interface Forum reconvened to correct errors and make clarifications in the MPI document of May 5, 1994, referred to below as Version 1.0. These discussions resulted in Version 1.1. The changes from Version 1.0 are minor. A version of this document with all changes marked is available.

¹⁴ Version 1.0: May, 1994. The Message-Passing Interface Forum (MPIF), with participation from over 40 organizations, has been meeting since January 1993 to discuss and define a set of library interface standards for message passing. MPIF is not sanctioned or supported by any official standards organization.

¹⁸ The goal of the Message-Passing Interface, simply stated, is to develop a widely used ¹⁹ standard for writing message-passing programs. As such the interface should establish a ²⁰ practical, portable, efficient, and flexible standard for message-passing.

This is the final report, Version 1.0, of the Message-Passing Interface Forum. This document contains all the technical features proposed for the interface. This copy of the draft was processed by LATEX on May 5, 1994.

Contents

Li	st of	Figures	xix
Li	st of	Tables	xxi
A	cknov	wledgments	xiii
1	Intr	roduction to MPI	1
	1.1	Overview and Goals	1
	1.2	Background of MPI-1.0.	2
	1.3	Background of MPI-1.1, MPI-1.2, and MPI-2.0	2
	1.4	Background of MPI-1.3 and MPI-2.1	3
	1.5	Background of MPI-2.2	4
	1.6	Background of MPI-3.0	4
	1.7	Background of MPI-3.1	4
	1.8	Background of 2019 Draft Specification	5
	1.9	Who Should Use This Standard?	5
	1.10	What Platforms Are Targets for Implementation?	5
		What Is Included in the Standard?	5
	1.12	What Is Not Included in the Standard?	6
	1.13	Organization of This Document	6
2	MPI	Terms and Conventions	9
	2.1	Document Notation	9
	2.2	Naming Conventions	9
	2.3	Procedure Specification	10
	2.4	Semantic Terms	11
		2.4.1 MPI Operations	11
		2.4.2 MPI Procedures	13
		2.4.3 MPI Datatypes	15
	2.5	Data Types	16
		2.5.1 Opaque Objects	16
		2.5.2 Array Arguments	18
		2.5.3 State	18
		2.5.4 Named Constants	18
		2.5.5 Choice	19
		2.5.6 Absolute Addresses and Relative Address Displacements	20
		2.5.7 File Offsets	20
		2.5.8 Counts	20

	2.6	Langua	age Binding
		2.6.1	Deprecated and Removed Interfaces
		2.6.2	Fortran Binding Issues
		2.6.3	C Binding Issues
		2.6.4	Functions and Macros
	2.7	Proces	ses
	2.8	Error I	Handling
	2.9	Implen	nentation Issues
		2.9.1	Independence of Basic Runtime Routines
		2.9.2	Interaction with Signals
	2.10	Examp	les $\ldots \ldots 26$
3	Poir	nt-to-P	oint Communication 27
	3.1	Introdu	action
	3.2	Blockii	ng Send and Receive Operations
		3.2.1	Blocking Send
		3.2.2	Message Data
		3.2.3	Message Envelope
		3.2.4	Blocking Receive
		3.2.5	Return Status
		3.2.6	Passing MPI_STATUS_IGNORE for Status
		3.2.7	Send-Receive
	3.3	Data T	Sype Matching and Data Conversion 40
		3.3.1	Type Matching Rules
			Type MPI_CHARACTER
		3.3.2	Data Conversion
	3.4		unication Modes
	3.5		tics of Point-to-Point Communication
	3.6	Buffer	Allocation and Usage
		3.6.1	Model Implementation of Buffered Mode54
	3.7		ocking Communication
			Communication Request Objects
		3.7.2	Communication Initiation
		3.7.3	Communication Completion
4		3.7.4	Semantics of Nonblocking Communications
		3.7.5	Multiple Completions 67
		3.7.6	Non-Destructive Test of status
	3.8	Probe	and Cancel
		3.8.1	Probe
		3.8.2	Matching Probe
		3.8.3	Matched Receives
		3.8.4	Cancel
	3.9		ent Communication Requests
	3.10	Null P	rocesses

4	Par	titione	d Point-to-Point Communication 93
	4.1	Introd	$uction \dots \dots$
	4.2	Seman	tics of Partitioned Point-to-Point Communication
		4.2.1	Communication Initialization and Starting with Partitioning 96
		4.2.2	Communication Completion under Partitioning 100
		4.2.3	Semantics of Communications in Partitioned Mode 101
	4.3	Partiti	ioned Communication Examples
		4.3.1	Partition Communication with Threads/Tasks Using OpenMP 4.0
			and greater
		4.3.2	Send-only Partitioning Example with Tasks and OpenMP version 4.0
			and greater
		4.3.3	Send and Receive Partitioning Example with OpenMP version 4.0
			and greater
	_		
5		atypes	109
	5.1		d Datatypes
		5.1.1	Type Constructors with Explicit Addresses
		5.1.2	Datatype Constructors
		5.1.3	Subarray Datatype Constructor
		5.1.4	Distributed Array Datatype Constructor
		5.1.5	Address and Size Functions
		5.1.6	Lower-Bound and Upper-Bound Markers
		5.1.7	Extent and Bounds of Datatypes
		5.1.8	True Extent of Datatypes
		5.1.9	Commit and Free
			Duplicating a Datatype 139
			Use of General Datatypes in Communication
			Correct Use of Addresses
			Decoding a Datatype
	- 0	5.1.14	Examples
	5.2		Ind Unpack
	5.3	Canon	ical MPI_PACK and MPI_UNPACK 166
6	Col	lective	Communication 171
			uction and Overview
	6.2		unicator Argument
		6.2.1	Specifics for Intracommunicator Collective Operations
		6.2.2	
		6.2.3	Specifics for Intercommunicator Collective Operations
	6.3		r Synchronization
	6.4		cast
		6.4.1	Example using MPI_BCAST
	6.5		\mathbf{r}
		6.5.1	Examples using MPI_GATHER, MPI_GATHERV
	6.6	Scatte	\mathbf{r}
		6.6.1	Examples using MPI_SCATTER, MPI_SCATTERV
	6.7	Gathe	r-to-all
		6.7.1	Example using MPI_ALLGATHER

6.8	All-to-	All Scatter/Gather	198
6.9	Global	Reduction Operations	204
	6.9.1	Reduce	204
	6.9.2	Predefined Reduction Operations	206
	6.9.3	Signed Characters and Reductions	209
	6.9.4	MINLOC and MAXLOC	209
	6.9.5	User-Defined Reduction Operations	213
		Example of User-Defined Reduce	216
	6.9.6	All-Reduce	218
	6.9.7	Process-Local Reduction	219
6.10	Reduce	e-Scatter	221
		MPI_REDUCE_SCATTER_BLOCK	221
		MPI_REDUCE_SCATTER	222
6.11	Scan		224
	6.11.1	Inclusive Scan	224
	6.11.2	Exclusive Scan	225
	6.11.3	Example using MPI_SCAN	226
6.12	Nonblo	ocking Collective Operations	228
		Nonblocking Barrier Synchronization	230
		Nonblocking Broadcast	230
		Example using MPI_IBCAST	231
	6.12.3	Nonblocking Gather	232
	6.12.4	Nonblocking Scatter	234
	6.12.5	Nonblocking Gather-to-all	236
		Nonblocking All-to-All Scatter/Gather	238
	6.12.7	Nonblocking Reduce	241
	6.12.8	Nonblocking All-Reduce	242
		Nonblocking Reduce-Scatter with Equal Blocks	243
	6.12.10	Nonblocking Reduce-Scatter	244
		Nonblocking Inclusive Scan	245
		Nonblocking Exclusive Scan	246
6.13	Persist	ent Collective Operations	246
	6.13.1	Persistent Barrier Synchronization	248
	6.13.2	Persistent Broadcast	249
	6.13.3	Persistent Gather	250
	6.13.4	Persistent Scatter	252
	6.13.5	Persistent Gather-to-all	254
	6.13.6	Persistent All-to-All Scatter/Gather	256
		Persistent Reduce	260
	6.13.8	Persistent All-Reduce	261
	6.13.9	Persistent Reduce-Scatter with Equal Blocks	262
		Persistent Reduce-Scatter	263
	6.13.11	Persistent Inclusive Scan	264
		Persistent Exclusive Scan	265
6.14	Correc	tness	265

7	Gro	oups, C	Contexts, Communicators, and Caching	275
	7.1	Introd	luction	275
		7.1.1	Features Needed to Support Libraries	275
		7.1.2	MPI's Support for Libraries	276
	7.2	Basic	Concepts	278
		7.2.1	Groups	278
		7.2.2	Contexts	278
		7.2.3	Intra-Communicators	279
		7.2.4	Predefined Intra-Communicators	279
	7.3	Group	Management	280
		7.3.1	Group Accessors	280
		7.3.2	Group Constructors	282
		7.3.3	Group Destructors	288
	7.4	Comm	nunicator Management	289
		7.4.1	Communicator Accessors	289
		7.4.2	Communicator Constructors	291
		7.4.3	Communicator Destructors	
		7.4.4	Communicator Info	308
	7.5	Motiva	ating Examples	311
		7.5.1	Current Practice #1	
		7.5.2	Current Practice $#2$	
		7.5.3	(Approximate) Current Practice #3	
		7.5.4	Example #4 \ldots	313
		7.5.5	Library Example $\#1$	314
		7.5.6	Library Example $#2$	316
	7.6		Communication	318
		7.6.1	Inter-Communicator Accessors	
		7.6.2	Inter-Communicator Operations	
		7.6.3	Inter-Communication Examples	
			Example 1: Three-Group "Pipeline"	
			Example 2: Three-Group "Ring"	
	7.7		ng	
		7.7.1	Functionality	
		7.7.2	Communicators	330
		7.7.3	Windows	336
		7.7.4	Datatypes	339
		7.7.5	Error Class for Invalid Keyval	343
	70	7.7.6	Attributes Example	343
	7.8		ng Objects	345
	7.9		lizing the Loosely Synchronous Model	350
		7.9.1	Basic Statements	350
		7.9.2	Models of Execution	$350 \\ 250$
			Static Communicator Allocation	350
			Dynamic Communicator Allocation	351
			The General Case	351

8	Pro	cess Topologies	353
	8.1	Introduction	353
	8.2	Virtual Topologies	354
	8.3	Embedding in MPI	354
	8.4	Overview of the Functions	354
	8.5	Topology Constructors	356
		8.5.1 Cartesian Constructor	356
		8.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE	357
		8.5.3 Graph Constructor	358
		8.5.4 Distributed Graph Constructor	360
		8.5.5 Topology Inquiry Functions	367
		8.5.6 Cartesian Shift Coordinates	375
		8.5.7 Partitioning of Cartesian Structures	377
		8.5.8 Low-Level Topology Functions	378
	8.6	Neighborhood Collective Communication	379
		8.6.1 Neighborhood Gather	380
		8.6.2 Neighbor Alltoall	384
	8.7	Nonblocking Neighborhood Communication	391
		8.7.1 Nonblocking Neighborhood Gather	
		8.7.2 Nonblocking Neighborhood Alltoall	393
	8.8	Persistent Neighborhood Communication	396
		8.8.1 Persistent Neighborhood Gather	396
		8.8.2 Persistent Neighborhood Alltoall	
	8.9	An Application Example	402
9	MPI	Environmental Management	407
	9.1	Implementation Information	
		9.1.1 Version Inquiries	407
		9.1.2 Environmental Inquiries	
		Tag Values	409
		Host Rank	409
		IO Rank	
		Clock Synchronization	410
		Inquire Processor Name	410
	9.2	Memory Allocation	411
	9.3	Error Handling	414
		9.3.1 Error Handlers for Communicators	417
		9.3.2 Error Handlers for Windows	419
		9.3.3 Error Handlers for Files	421
		9.3.4 Error Handlers for Sessions	422
		9.3.5 Freeing Errorhandlers and Retrieving Error Strings	424
	9.4	Error Codes and Classes	425
	9.5	Error Classes, Error Codes, and Error Handlers	426
	9.6	Timers and Synchronization	433
10	The	e Info Object	435

х

11 Process Initialization, Creation, and Management		443
11.1 Introduction		443
11.2 The World Model		444
11.2.1 Starting MPI Processes		444
11.2.2 Finalizing MPI		450
11.2.3 Determining Whether MPI Has Been Initialized When Using the V	World	1
Model		
11.2.4 Allowing User Functions at MPI Finalization		
11.3 The Sessions Model		
11.3.1 Session Creation and Destruction Methods		
11.3.2 Processes Sets		
11.3.3 Runtime Query Functions		
11.3.4 Sessions Model Examples		
11.4 Common Elements of Both Process Models		468
11.4.1 MPI Functionality that is Always Available		468
11.4.2 Aborting MPI Processes		469
11.5 Portable MPI Process Startup		
11.6 MPI and Threads		
11.6.1 General		
11.6.2 Clarifications		
11.7 The Dynamic Process Model		
11.7.1 Starting Processes		
11.7.2 The Runtime Environment	• • •	476
11.8 Process Manager Interface		477
11.8.1 Processes in MPI		477
11.8.2 Starting Processes and Establishing Communication		
11.8.3 Starting Multiple Executables and Establishing Communication		
11.8.4 Reserved Keys		
11.8.5 Spawn Example		486
11.9 Establishing Communication		
11.9.1 Names, Addresses, Ports, and All That		
11.9.2 Server Routines		
11.9.3 Client Routines		
11.9.4 Name Publishing		493
11.9.5 Reserved Key Values		495
11.9.6 Client/Server Examples		496
11.10Other Functionality		498
11.10.1 Universe Size \ldots		498
11.10.2 Singleton MPI Initialization		499
11.10.3 MPI_APPNUM		500
11.10.4 Releasing Connections		500
11.10.5 Another Way to Establish MPI Communication		
12 One-Sided Communications		505
12.1 Introduction		505
12.2 Initialization		506
12.2.1 Window Creation		507
12.2.2 Window That Allocates Memory		

	12.2.3 Window That Allocates Shared Memory
	12.2.4 Window of Dynamically Attached Memory 51
	12.2.5 Window Destruction
	12.2.6 Window Attributes
	12.2.7 Window Info
12.3	Communication Calls
	12.3.1 Put
	12.3.2 Get
	12.3.3 Examples for Communication Calls
	12.3.4 Accumulate Functions
	Accumulate Function
	Get Accumulate Function
	Fetch and Op Function 53
	Compare and Swap Function
	12.3.5 Request-based RMA Communication Operations
12.4	Memory Model
	Synchronization Calls
	12.5.1 Fence
	12.5.2 General Active Target Synchronization
	12.5.3 Lock
	12.5.4 Flush and Sync
	12.5.5 Assertions
	12.5.6 Miscellaneous Clarifications
12.6	Error Handling
	12.6.1 Error Handlers
	12.6.2 Error Classes
12.7	Semantics and Correctness
	12.7.1 Atomicity
	12.7.2 Ordering
	12.7.3 Progress
	12.7.4 Registers and Compiler Optimizations
12.8	Examples
	ernal Interfaces 583
	Introduction
13.2	Generalized Requests
	13.2.1 Examples
13.3	Associating Information with Status
14 I/O	593
	Introduction
14.1	14.1.1 Definitions 59
1/1 9	File Manipulation
14.4	14.2.1 Opening a File 59
	14.2.1 Opening a File 59 14.2.2 Closing a File 59
	14.2.2 Closing a File
	14.2.4 Resizing a File
	14.2.4 Resizing a File
	14.2.0 Treatocating space for a file

	14.2.6	Querying the Size of a File	599
	14.2.7	Querying File Parameters	599
	14.2.8	File Info	601
		Reserved File Hints	602
14.3	File Vi	iews	604
14.4	Data A	Access	607
	14.4.1	Data Access Routines	607
		Positioning	608
		Synchronism	608
		Coordination	609
		Data Access Conventions	609
	14.4.2	Data Access with Explicit Offsets	610
	14.4.3	Data Access with Individual File Pointers	615
	14.4.4	Data Access with Shared File Pointers	623
		Noncollective Operations	624
		Collective Operations	626
		Seek	628
	14.4.5	Split Collective Data Access Routines	630
14.5	File In	teroperability	637
	14.5.1	Datatypes for File Interoperability	639
	14.5.2	External Data Representation: "external32"	641
	14.5.3	User-Defined Data Representations	644
		Extent Callback	645
		Datarep Conversion Functions	645
	14.5.4	Matching Data Representations	647
14.6	Consis	tency and Semantics	648
	14.6.1	File Consistency	648
	14.6.2	Random Access vs. Sequential Files	651
	14.6.3	Progress	652
	14.6.4	Collective File Operations	652
	14.6.5	Nonblocking Collective File Operations	652
	14.6.6	Type Matching	653
	14.6.7	Miscellaneous Clarifications	653
	14.6.8	MPI_Offset Type	653
		Logical vs. Physical File Layout	653
)File Size	654
	14.6.11	Examples	654
		Asynchronous I/O	657
		rror Handling	658
14.8	I/O Er	rror Classes	659
14.9	Examp		659
		Double Buffering with Split Collective I/O	659
	14.9.2	Subarray Filetype Constructor	662

15 Tool	Supp	ort	665
15.1	Introdu	uction	665
15.2	Profili	ng Interface	665
	15.2.1	Requirements	665
	15.2.2	Discussion	666
	15.2.3	Logic of the Design	666
		Miscellaneous Control of Profiling	667
		Profiler Implementation Example	668
		MPI Library Implementation Example	668
		Systems with Weak Symbols	668
		Systems without Weak Symbols	669
	15.2.7	Complications	669
		Multiple Counting	669
		Linker Oddities	670
		Fortran Support Methods	670
	15.2.8	Multiple Levels of Interception	670
15.3	The M	PI Tool Information Interface	671
	15.3.1	Verbosity Levels	672
	15.3.2	Binding MPI Tool Information Interface Variables to MPI Objects .	672
	15.3.3	Convention for Returning Strings	673
	15.3.4	Initialization and Finalization	674
		Datatype System	675
	15.3.6	Control Variables	678
		Control Variable Query Functions	678
		Example: Printing All Control Variables	681
		Handle Allocation and Deallocation	682
		Control Variable Access Functions	683
		Example: Reading the Value of a Control Variable	684
	15.3.7	Performance Variables	685
		Performance Variable Classes	685
		Performance Variable Query Functions	687
		Performance Experiment Sessions	690
		Handle Allocation and Deallocation	691
		Starting and Stopping of Performance Variables	692
		Performance Variable Access Functions	693
		Example: Tool to Detect Receives with Long Unexpected Message	
		Queues	695
	15.3.8	Events	697
		Event Sources	698
		Callback Safety Requirements	700
		Event Type Query Functions	702
		Handle Allocation and Deallocation	705
		Handling Dropped Events	709
		Reading Event Data	711
		Reading Event Meta Data	712
	15.3.9	Variable Categorization	713
		Category Query Functions	714
		Category Member Query Functions	716

	15.3.10 Return Codes for the MPI Tool Information Interface	718
	15.3.11 Profiling Interface	719
16 Der	precated Interfaces	721
	Deprecated since MPI-2.0	721
	Deprecated since MPI-2.2	724
	Deprecated since MPI-3.2	724
	Deprecated since MPI-4.0	724
17 Dor	noved Interfaces	727
	Removed MPI-1 Bindings	727
11.1	17.1.1 Overview	727
	17.1.2 Removed MPI-1 Functions	727
	17.1.2 Removed MPI-1 Datatypes	727
	17.1.4 Removed MPI-1 Constants	727
	17.1.4 Removed MPI-1 Collback Prototypes	728
17.9	17.1.5 Removed MPI-1 Callback Prototypes	728
11.2	C++ bindings	120
	kward Incompatibilities	729
18.1	Backward Incompatible since MPI-3.2	729
19 Lan	guage Bindings	731
19.1	Fortran Support	731
	19.1.1 Overview	731
	19.1.2 Fortran Support Through the mpi_f08 Module	732
	19.1.3 Fortran Support Through the mpi Module	735
	19.1.4 Fortran Support Through the mpif.h Include File	737
	19.1.5 Interface Specifications, Procedure Names, and the Profiling Interface	738
	19.1.6 MPI for Different Fortran Standard Versions	743
	19.1.7 Requirements on Fortran Compilers	747
	19.1.8 Additional Support for Fortran Register-Memory-Synchronization	748
	19.1.9 Additional Support for Fortran Numeric Intrinsic Types	749
	Parameterized Datatypes with Specified Precision and Exponent Range	<mark>e</mark> 750
	Support for Size-specific MPI Datatypes	754
	Communication With Size-specific Types	756
	19.1.10 Problems With Fortran Bindings for MPI	757
	19.1.11 Problems Due to Strong Typing	759
	19.1.12 Problems Due to Data Copying and Sequence Association with Sub-	
	script Triplets	759
	19.1.13 Problems Due to Data Copying and Sequence Association with Vector	
	Subscripts	762
	19.1.14 Special Constants	763
	19.1.15 Fortran Derived Types	763
	19.1.16 Optimization Problems, an Overview	765
	19.1.17 Problems with Code Movement and Register Optimization	766
	Nonblocking Operations	766
	Persistent Operations	767
	One-sided Communication	767

		MPI_BOTTOM and Combining Independent Variables in Datatypes	. 767
		Solutions	. 768
		The Fortran ASYNCHRONOUS Attribute	. 768
		Calling MPI_F_SYNC_REG	. 770
		A User Defined Routine Instead of MPI_F_SYNC_REG	. 771
		Module Variables and COMMON Blocks	. 771
		The (Poorly Performing) Fortran VOLATILE Attribute	. 772
		The Fortran TARGET Attribute	
	19.1.18	8 Temporary Data Movement and Temporary Memory Modification	
		Permanent Data Movement	
		Comparison with C	
19.2		age Interoperability	
10.2		Introduction	
		Assumptions	
	10.2.2	Initialization	. 779
	10.2.0	Transfer of Handles	. 780
		Status	
	19.2.0	MPL Opeque Objecta	. 782
	19.2.0	MPI Opaque Objects	
		Datatypes	
		Callback Functions	
		Error Handlers	
	10.0 7	Reduce Operations	
		Attributes	
		Extra-State	
	1000		
		Constants	
		Constants	
A Low	19.2.1	Interlanguage Communication	. 792
A Lan	19.2.1	Interlanguage Communication	. 792
A La n A.1	19.2.10 Iguage Define	Dinterlanguage Communication	. 792 795 . 795
A Lan A.1	19.2.10 Iguage Define A.1.1	Dinterlanguage Communication	. 792 795 . 795 . 795
A La n A.1	19.2.10 iguage Define A.1.1 A.1.2	Dinterlanguage Communication Bindings Summary d Values and Handles Defined Constants Types	. 792 795 . 795 . 795 . 809
A Lan A.1	19.2.10 iguage Define A.1.1 A.1.2	Dinterlanguage Communication Bindings Summary d Values and Handles Defined Constants Types Prototype Definitions	. 792 795 . 795 . 795 . 809 . 810
A Lan A.1	19.2.10 iguage Define A.1.1 A.1.2	Bindings Summary d Values and Handles Defined Constants Types Prototype Definitions C Bindings	. 792 795 . 795 . 795 . 809 . 810 . 810
A Lan A.1	19.2.10 iguage Define A.1.1 A.1.2	Bindings Summary d Values and Handles Defined Constants Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module	. 792 795 . 795 . 795 . 809 . 810 . 810 . 811
A Lan A.1	19.2.10 nguage Define A.1.1 A.1.2 A.1.3	Bindings Summary d Values and Handles Defined Constants Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module Fortran Bindings with mpif.h or the mpi Module	. 792 795 . 795 . 795 . 809 . 810 . 810 . 811 . 814
A Lan A.1	19.2.10 guage Define A.1.1 A.1.2 A.1.3 A.1.4	Bindings Summary d Values and Handles Defined Constants Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module Fortran Bindings with mpif.h or the mpi Module Deprecated Prototype Definitions	. 792 795 . 795 . 795 . 809 . 810 . 810 . 811 . 814 . 814 . 816
A Lan A.1	19.2.10 nguage Define A.1.1 A.1.2 A.1.3 A.1.4 A.1.5	Bindings Summary d Values and Handles Defined Constants Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module Fortran Bindings with mpif.h or the mpi Module Deprecated Prototype Definitions Info Keys	 792 795 795 795 809 810 810 811 814 816 817
A Lan A.1	19.2.10 guage Define A.1.1 A.1.2 A.1.3 A.1.4 A.1.5 A.1.6	Bindings Summary d Values and Handles Defined Constants Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module Fortran Bindings with mpif.h or the mpi Module Info Keys Info Values	. 792 795 . 795 . 795 . 809 . 810 . 810 . 811 . 814 . 814 . 816 . 817 . 818
A Lan A.1	19.2.10 guage Define A.1.1 A.1.2 A.1.3 A.1.4 A.1.5 A.1.6 Summ	Bindings Summary d Values and Handles Defined Constants Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module Fortran Bindings with mpif.h or the mpi Module Deprecated Prototype Definitions Info Keys Info Values ary of the Semantics of all OpRelated Routines	 792 795 795 795 809 810 810 811 814 816 817 818 818
A.1	19.2.10 guage Define A.1.1 A.1.2 A.1.3 A.1.4 A.1.5 A.1.6 Summ C Bine	Bindings Summary d Values and Handles Defined Constants Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module Fortran Bindings with mpif.h or the mpi Module Deprecated Prototype Definitions Info Keys Info Values ary of the Semantics of all OpRelated Routines	 792 795 795 795 809 810 810 811 814 816 817 818 818 819
A.1	19.2.10 guage Define A.1.1 A.1.2 A.1.3 A.1.4 A.1.5 A.1.6 Summ C Bine A.3.1	Bindings Summary d Values and Handles Defined Constants Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module Fortran Bindings with mpif.h or the mpi Module Deprecated Prototype Definitions Info Keys Info Values ary of the Semantics of all OpRelated Routines Point-to-Point Communication C Bindings	 792 795 795 795 809 810 811 814 816 817 818 818 819 819 819
A.1	19.2.10 guage Define A.1.1 A.1.2 A.1.3 A.1.4 A.1.5 A.1.6 Summ C Bino A.3.1 A.3.2	Bindings Summary d Values and Handles Defined Constants Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module Fortran Bindings with mpif.h or the mpi Module Deprecated Prototype Definitions Info Keys Info Values ary of the Semantics of all OpRelated Routines Point-to-Point Communication C Bindings	 792 795 795 795 809 810 811 814 816 817 818 818 819 821
A.1	19.2.10 guage Define A.1.1 A.1.2 A.1.3 A.1.4 A.1.5 A.1.6 Summ C Bine A.3.1	Dinterlanguage Communication Bindings Summary d Values and Handles Defined Constants Types Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module Fortran Bindings with mpif.h or the mpi Module Deprecated Prototype Definitions Info Keys Info Values ary of the Semantics of all OpRelated Routines Point-to-Point Communication C Bindings Partitioned Communication C Bindings Datatypes C Bindings	 792 795 795 795 795 809 810 810 811 814 816 817 818 818 818 819 821 821
A.1	19.2.10 guage Define A.1.1 A.1.2 A.1.3 A.1.4 A.1.5 A.1.6 Summ C Bino A.3.1 A.3.2	Bindings Summary d Values and Handles Defined Constants Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module Fortran Bindings with mpif.h or the mpi Module Deprecated Prototype Definitions Info Keys Info Values ary of the Semantics of all OpRelated Routines Point-to-Point Communication C Bindings	 792 795 795 795 795 809 810 810 811 814 816 817 818 818 818 819 821 821
A.1	19.2.10 guage Define A.1.1 A.1.2 A.1.3 A.1.4 A.1.5 A.1.6 Summ C Bine A.3.1 A.3.2 A.3.3	Dinterlanguage Communication Bindings Summary d Values and Handles Defined Constants Types Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module Fortran Bindings with mpif.h or the mpi Module Deprecated Prototype Definitions Info Keys Info Values ary of the Semantics of all OpRelated Routines Point-to-Point Communication C Bindings Partitioned Communication C Bindings Datatypes C Bindings	 792 795 795 795 795 809 810 811 814 816 817 818 818 819 821 821 823
A.1	19.2.10 guage Define A.1.1 A.1.2 A.1.3 A.1.4 A.1.5 A.1.6 Summ C Bine A.3.1 A.3.2 A.3.3 A.3.4	Bindings Summary d Values and Handles Defined Constants Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module Fortran Bindings with mpif.h or the mpi Module Deprecated Prototype Definitions Info Keys Info Values ary of the Semantics of all OpRelated Routines Point-to-Point Communication C Bindings Partitioned Communication C Bindings Datatypes C Bindings	 792 795 795 795 795 809 810 811 814 816 817 818 818 819 821 823 827
A.1	19.2.10 guage Define A.1.1 A.1.2 A.1.3 A.1.4 A.1.5 A.1.6 Summ C Bine A.3.1 A.3.2 A.3.3 A.3.4 A.3.5	Bindings Summary d Values and Handles Defined Constants Types Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module Fortran Bindings with mpif.h or the mpi Module Deprecated Prototype Definitions Info Keys Info Values ary of the Semantics of all OpRelated Routines Point-to-Point Communication C Bindings Partitioned Communication C Bindings Datatypes C Bindings Groups, Contexts, Communicators, and Caching C Bindings	792 795 795 795 809 810 811 814 814 816 817 818 818 819 821 823 827 830
A.1	19.2.10 guage Define A.1.1 A.1.2 A.1.3 A.1.4 A.1.5 A.1.6 Summ C Bine A.3.1 A.3.2 A.3.3 A.3.4 A.3.5 A.3.6	Dinterlanguage Communication Bindings Summary d Values and Handles Defined Constants Types Types Prototype Definitions C Bindings Fortran 2008 Bindings with the mpi_f08 Module Fortran Bindings with mpif.h or the mpi Module Deprecated Prototype Definitions Info Keys Info Values ary of the Semantics of all OpRelated Routines dings Partitioned Communication C Bindings Datatypes C Bindings Collective Communication C Bindings Groups, Contexts, Communicators, and Caching C Bindings	. 792 795 . . 795 . 795 . 809 . 810 . 810 . 810 . 811 . 814 . 814 . 817 . 818 . 819 . 819 . 821 . 823 . 823 . 830 . 833

			Process Creation and Management C Bindings	834
			One-Sided Communications C Bindings	836
		A.3.11	External Interfaces C Bindings	838
			I/O C Bindings	
			Language Bindings C Bindings	
			Tools / Profiling Interface C Bindings	842
			Tools / MPI Tool Information Interface C Bindings	842
			Deprecated C Bindings	
	A.4		n 2008 Bindings with the mpi_f08 Module	
			Point-to-Point Communication Fortran 2008 Bindings	846
		A.4.2	0	852
			Datatypes Fortran 2008 Bindings	
			Collective Communication Fortran 2008 Bindings	
		A.4.5	Groups, Contexts, Communicators, and Caching Fortran 2008 Binding	
		A.4.6	Process Topologies Fortran 2008 Bindings	
		A.4.7	U	
		A.4.8	The Info Object Fortran 2008 Bindings	
			Process Creation and Management Fortran 2008 Bindings	
			One-Sided Communications Fortran 2008 Bindings	888
			External Interfaces Fortran 2008 Bindings	
			I/O Fortran 2008 Bindings	894
			Language Bindings Fortran 2008 Bindings	902
			Tools / Profiling Interface Fortran 2008 Bindings	903
	A =		Deprecated Fortran 2008 Bindings	
	A.5		n Bindings with mpif.h or the mpi Module	904
			Point-to-Point Communication Fortran Bindings	904
			Partitioned Communication Fortran Bindings	907 907
		A.5.4	Collective Communication Fortran Bindings	907 910
		A.5.4 A.5.5	Groups, Contexts, Communicators, and Caching Fortran Bindings	910 916
		A.5.6	Process Topologies Fortran Bindings	910 920
		A.5.7		920 924
			The Info Object Fortran Bindings	924 926
			Process Creation and Management Fortran Bindings	927
	\frown		One-Sided Communications Fortran Bindings	928
			External Interfaces Fortran Bindings	933
			I/O Fortran Bindings	933
			Language Bindings Fortran Bindings	938
			Tools / Profiling Interface Fortran Bindings	938
			Deprecated Fortran Bindings	938
			T	
Β		nge-Lo		941
	B.1	U U	es from Version 3.2 to Version 4.0	941
		B.1.1	Fixes to Errata in Previous Versions of MPI	941
	_	B.1.2	Changes in MPI-4.0	941
	B.2		es from Version 3.1 to Version 3.2	942
		B.2.1	Fixes to Errata in Previous Versions of MPI	942
		B.2.2	Changes in MPI-3.2	942

B.3	Changes from Version 3.0 to Version 3.1	944	
	B.3.1 Fixes to Errata in Previous Versions of MPI	944	
	B.3.2 Changes in MPI-3.1	946	
B.4	Changes from Version 2.2 to Version 3.0	946	
	B.4.1 Fixes to Errata in Previous Versions of MPI	946	
	B.4.2 Changes in MPI-3.0	947	
B.5	Changes from Version 2.1 to Version 2.2	952	
B.6	Changes from Version 2.0 to Version 2.1	955	
Bibliog	raphy	961	
Genera	l Index	967	
Examp	les Index	973	
MPI C	onstant and Predefined Handle Index	976	
MPI D	eclarations Index	982	
MPI C	allback Function Prototype Index	983	
MPI F	MPI Function Index		

List of Figures

	2.1	State Transition Diagram for Blocking Operations	12
	2.2	State Transition Diagram for Nonblocking Operations	12
	2.3	State Transition Diagram for Persistent Operations	13
	6.1	Collective communications, an overview	173
	6.2	Intercommunicator allgather	176
	6.3	Intercommunicator reduce-scatter	177
	6.4	Gather example	183
	6.5	Gatherv example with strides	184
	6.6	Gatherv example, 2-dimensional	185
	6.7	Gatherv example, 2-dimensional, subarrays with different sizes	186
	6.8	Gatherv example, 2-dimensional, subarrays with different sizes and strides .	188
	6.9	Scatter example	193
	6.10	Scatterv example with strides	193
	6.11	Scatterv example with different strides and counts	195
	6.12	Race conditions with point-to-point and collective communications	268
	6.13	Overlapping Communicators Example	272
	7.1	Intercommunicator creation using MPI_COMM_CREATE	297
	7.2	Intercommunicator construction with MPI_COMM_SPLIT	301
	7.3	Recursive communicator creation with MPI_COMM_SPLIT_TYPE	307
	7.4	Three-group pipeline	326
	7.5	Three-group ring	327
	8.1	Neighborhood gather communication example.	382
4	8.2	Cartesian neighborhood allgather example for 3 and 1 processes in a dimension.	.383
	8.3	Cartesian neighborhood alltoall example for 3 and 1 processes in a dimension.	386
	8.4	Set-up of process structure for two-dimensional parallel Poisson solver	403
	8.5	Communication routine with local data copying and sparse neighborhood all-to-all.	404
	8.6	Communication routine with sparse neighborhood all-to-all-w and without	
		local data copying.	405
	8.7	Two-dimensional parallel Poisson solver with persistent sparse neighborhood	
		all-to-all-w and without local data copying.	406
	11.1	Session handle to communicator	456
	11.2	Process set examples	459

12.1	Schematic description of the public/private window operations in the	
	MPI_WIN_SEPARATE memory model for two overlapping windows	541
12.2	Active target communication	544
12.3	Active target communication, with weak synchronization	545
12.4	Passive target communication	546
12.5	Active target communication with several processes	549
12.6	Symmetric communication	569
12.7	Deadlock situation	569
12.8	No deadlock	569
	Etypes and filetypes	592
14.2	Partitioning a file among parallel processes	592
14.3	Displacements	605
14.4	Example array file layout	662
14.5	Example local array filetype for process 1	662
19.1	Status conversion routines	783

List of Tables

2.1	Deprecated and Removed constructs	22
$3.1 \\ 3.2 \\ 3.3 \\ 3.4$	Predefined MPI datatypes corresponding to Fortran datatypes Predefined MPI datatypes corresponding to C datatypes Predefined MPI datatypes corresponding to both C and Fortran datatypes Predefined MPI datatypes corresponding to C++ datatypes	29 30 31 31
5.1	combiner values returned from MPI_TYPE_GET_ENVELOPE	145
7.1	MPI_COMM_* Function Behavior (in Inter-Communication Mode)	320
9.1 9.2	Error classes (Part 1)	427 428
11.1	List of MPI Functions that can be called at any time within an MPI program, including prior to MPI initialization and following MPI finalization	468
	C types of attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR	519 559
$14.2 \\ 14.3 \\ 14.4$	Data access routines	607 642 643 643 660
$15.1 \\ 15.2 \\ 15.3 \\ 15.4 \\ 15.5$	MPI tool information interface verbosity levels	673 673 676 680 700
	List of MPI functions that when called from within a callback function may not return MPI_T_ERR_NOT_ACCESSIBLE	701 720
$\begin{array}{c} 17.2 \\ 17.3 \end{array}$	Removed MPI-1 functions and their replacementsRemoved MPI-1 datatypes and their replacementsRemoved MPI-1 constantsRemoved MPI-1 callback prototypes and their replacements	727 728 728 728 728

1 2	19.1 Specific Fortran procedure names and related calling conventions73919.2 Occurrence of Fortran optimization problems765
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24	
25 26	
20 27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

Acknowledgments

This document is the product of a number of distinct efforts in four distinct phases: one for each of MPI-1, MPI-2, MPI-3, and MPI-4. This section describes these in historical order, starting with MPI-1. Some efforts, particularly parts of MPI-2, had distinct groups of individuals associated with them, and these efforts are detailed separately.

This document represents the work of many people who have served on the MPI Forum. The meetings have been attended by dozens of people from many parts of the world. It is the hard and dedicated work of this group that has led to the MPI standard. The technical development was carried out by subgroups, whose work was reviewed by the full committee. During the period of development of the Message-Passing Interface (MPI), many people helped with this effort. Those who served as primary coordinators in MPI-1.0 and MPI-1.1 are: • Jack Dongarra, David Walker, Conveners and Meeting Chairs • Ewing Lusk, Bob Knighten, Minutes • Marc Snir, William Gropp, Ewing Lusk, Point-to-Point Communication 24 • Al Geist, Marc Snir, Steve Otto, Collective Communication • Steve Otto, Editor • Rolf Hempel, Process Topologies • Ewing Lusk, Language Binding • William Gropp, Environmental Management • James Cownie, Profiling • Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Mark Sears, Groups, Contexts, and Communicators • Steven Huss-Lederman, Initial Implementation Subset The following list includes some of the active participants in the MPI-1.0 and MPI-1.1 process not mentioned above.

xxiii

 $\overline{7}$

1	Ed Anderson	Robert Babb	Joe Baron	Eric Barszcz
2	Scott Berryman	Rob Bjornson	Nathan Doss	Anne Elster
$\frac{3}{4}$	Jim Feeney Daniel Frye	Vince Fernando Ian Glendinning	Sam Fineberg Adam Greenberg	Jon Flower Robert Harrison
4 5	Leslie Hart	Tom Haupt	Don Heller	Tom Henderson
6	Alex Ho	C.T. Howard Ho	Gary Howell	John Kapenga
7	James Kohl	Susan Krauss	Bob Leary	Arthur Maccabe
8	Peter Madams	Alan Mainwaring	Oliver McBryan	Phil McKinley
9	Charles Mosher	Dan Nessett	Peter Pacheco	Howard Palmer
10	Paul Pierce	Sanjay Ranka	Peter Rigsbee	Arch Robison
11	Erich Schikuta	Ambuj Singh	Alan Sussman	Robert Tomlinson
12	Robert G. Voigt	Dennis Weeks	Stephen Wheat	Steve Zenith
13			1 NT	
14	-		-	atory made the draft avail-
15				distributing the document. RPA and NSF under grant
16		-		nology Center Cooperative
17 18				opean Community through
19	Esprit project P6643 (Pl			
20	(
21				
22	MPI-1.2 and MPI-2.0:			
23	Those who served as prin	mary coordinators in	n MPI-1.2 and MPI-	2.0 are:
24				
25 26	• Ewing Lusk, Conve	ener and Meeting Cl	nair	
27	• Steve Huss-Lederm	an, Editor		
28 29	• Ewing Lusk, Misce	llany		
30	• Bill Saphir, Proces	s Creation and Man	agement	
31 32	• Marc Snir, One-Sid	led Communications	3	
33	• Bill Gropp and An	thony Skjellum, Ext	ended Collective O	perations
34 35	• Steve Huss-Lederm	an, External Interfa	ices	
36	• Bill Nitzberg, I/O			
37 38	• Andrew Lumsdaine	e, Bill Saphir, and J	eff Squyres, Langua	ge Bindings
39 40	• Anthony Skjellum	and Arkady Kaneys	ky. Real-Time	
40				
42	The following list in	cludes some of the a	active participants w	who attended MPI-2 Forum
43	meetings and are not me	entioned above.		
44				
45				
46				
47				
48				

Greg Astfalk	Robert Babb	Ed Benson	Rajesh Bordawekar	1	
Pete Bradley	Peter Brennan	Ron Brightwell	Maciej Brodowicz	2	
Eric Brunner	Greg Burns	Margaret Cahir	Pang Chen	3	
Ying Chen	Albert Cheng	Yong Cho	Joel Clark	4	
Lyndon Clarke	Laurie Costello	Dennis Cottel	Jim Cownie	5	
Zhenqian Cui	Suresh Damodaran-Kar	nal	Raja Daoud	6	
Judith Devaney	David DiNucci	Doug Doefler	Jack Dongarra	7	
Terry Dontje	Nathan Doss	Anne Elster	Mark Fallon	8	
Karl Feind	Sam Fineberg	Craig Fischberg	Stephen Fleischman	9	
Ian Foster	Hubertus Franke	Richard Frost	Al Geist	10	
Robert George	David Greenberg	John Hagedorn	Kei Harada	11	
Leslie Hart	Shane Hebert	Rolf Hempel	Tom Henderson	12	
Alex Ho	Hans-Christian Hoppe	Joefon Jann	Terry Jones	13	
Karl Kesselman	Koichi Konishi	Susan Kraus	Steve Kubica	14	
Steve Landherr	Mario Lauria	Mark Law	Juan Leon	15	
Lloyd Lewins	Ziyang Lu	Bob Madahar	Peter Madams	16	
John May	Oliver McBryan	Brian McCandless	Tyce McLarty	17	
Thom McMahon	Harish Nag	Nick Nevin	Jarek Nieplocha	18	
Ron Oldfield	Peter Ossadnik	Steve Otto	Peter Pacheco	19	
Yoonho Park	Perry Partow	Pratap Pattnaik	Elsie Pierce	20	
Paul Pierce	Heidi Poxon	Jean-Pierre Prost	Boris Protopopov	21	
James Pruyve	Rolf Rabenseifner	Joe Rieken	Peter Rigsbee	22	
Tom Robey	Anna Rounbehler	Nobutoshi Sagawa	Arindam Saha	23	
Eric Salo	Darren Sanders	Eric Sharakan	Andrew Sherman	24	
Fred Shirley	Lance Shuler	A. Gordon Smith	Ian Stockdale	25	
David Taylor	Stephen Taylor	Greg Tensa	Rajeev Thakur	26	
Marydell Tholburn	Dick Treumann	Simon Tsang	Manuel Ujaldon	27	
David Walker	Jerrell Watts	Klaus Wolf	Parkson Wong	28	
Dave Wright				29	
				30	
The MPI Forum a	The MPI Forum also acknowledges and appreciates the valuable input from people via $_{31}$				

The MPI Forum also acknowledges and appreciates the valuable input from people via e-mail and in person.

The following institutions supported the MPI-2 effort through time and travel support for the people listed above.

Argonne National Laboratory	35
Bolt, Beranek, and Newman	36
California Institute of Technology	37
Center for Computing Sciences	38
Convex Computer Corporation	39
Cray Research	40
Digital Equipment Corporation	41
Dolphin Interconnect Solutions, Inc.	42
Edinburgh Parallel Computing Centre	43
General Electric Company	44
German National Research Center for Information Technology	45
Hewlett-Packard	46
Hitachi	47
Hughes Aircraft Company	48

32

33

1	Intel Corporation
2	International Business Machines
3	Khoral Research
4	Lawrence Livermore National Laboratory
5	Los Alamos National Laboratory
6	MPI Software Techology, Inc.
7	Mississippi State University
8	NEC Corporation
9	National Aeronautics and Space Administration
10	National Energy Research Scientific Computing Center
11	National Institute of Standards and Technology
12	National Oceanic and Atmospheric Adminstration
13	Oak Ridge National Laboratory
14	The Ohio State University
15	PALLAS GmbH
16	Pacific Northwest National Laboratory
17	Pratt & Whitney
18	San Diego Supercomputer Center
19	Sanders, A Lockheed-Martin Company
20	Sandia National Laboratories
21	Schlumberger
22	Scientific Computing Associates, Inc.
23	Silicon Graphics Incorporated
24	Sky Computers
25	Sun Microsystems Computer Corporation
26	Syracuse University
27	The MITRE Corporation
28	Thinking Machines Corporation
29	United States Navy
30	University of Colorado
31	University of Denver
32	University of Houston
33	University of Illinois
34	University of Maryland
35	University of Notre Dame
36	University of San Fransisco
37	University of Stuttgart Computing Center
38	University of Wisconsin
39	
40	MPI-2 operated on a very tight budget (in reality, it had no budget when the first
41	meeting was announced). Many institutions helped the MPI-2 effort by supporting the
42	efforts and travel of the members of the MPI Forum. Direct support was given by NSF and
43	DARPA under NSF contract CDA-9115428 for travel by U.S. academic participants and
44	Esprit under project HPC Standards (21111) for European participants.
45	
46	
47	
48	

MPI-1.3 and MPI-2.1:	1
The editors and organizers of the combined documents have been:	2 3
• Richard Graham, Convener and Meeting Chair	4
• Jack Dongarra, Steering Committee	5 6
• Al Geist, Steering Committee	7
	8 9
• Bill Gropp, Steering Committee	10
• Rainer Keller, Merge of MPI-1.3	11
• Andrew Lumsdaine, Steering Committee	12 13
• Ewing Lusk, Steering Committee, MPI-1.1-Errata (Oct. 12, 1998) MPI-2.1-Errata Ballots 1, 2 (May 15, 2002)	14 15
• Rolf Rabenseifner, Steering Committee, Merge of MPI-2.1 and MPI-2.1-Errata Ballots 3, 4 (2008)	16 17 18
All chapters have been revisited to achieve a consistent MPI-2.1 text. Those who served as authors for the necessary modifications are:	19 20 21
• Bill Gropp, Front matter, Introduction, and Bibliography	22
• Richard Graham, Point-to-Point Communication	23 24
• Adam Moody, Collective Communication	25 26
• Richard Treumann, Groups, Contexts, and Communicators	27
• Jesper Larsson Träff, Process Topologies, Info-Object, and One-Sided Communica- tions	28 29 30
• George Bosilca, Environmental Management	31 32
• David Solt, Process Creation and Management	33
• Bronis R. de Supinski, External Interfaces, and Profiling	34
	35 36
• Rajeev Thakur, I/O	37
• Jeffrey M. Squyres, Language Bindings and MPI-2.1 Secretary	38 39
• Rolf Rabenseifner, Deprecated Functions and Annex Change-Log	40
• Alexander Supalov and Denis Nagorny, Annex Language Bindings	41 42
The following list includes some of the active participants who attended MPI-2 Forum	43
meetings and in the e-mail discussions of the errata items and are not mentioned above.	44 45
	45 46
	47
	48

1	Pavan Balaji	Purushotham V. Bangalore	Brian Barrett
2	Richard Barrett	Christian Bell	Robert Blackmore
3	Gil Bloch	Ron Brightwell	Jeffrey Brown
4	Darius Buntinas	Jonathan Carter	Nathan DeBardeleben
5	Terry Dontje	Gabor Dozsa	Edric Ellis
6	Karl Feind	Edgar Gabriel	Patrick Geoffray
7	David Gingold	Dave Goodell	Erez Haba
8	Robert Harrison	Thomas Herault	Steve Hodson
9	Torsten Hoefler	Joshua Hursey	Yann Kalemkarian
10	Matthew Koop	Quincey Koziol	Sameer Kumar
11	Miron Livny	Kannan Narasimhan	Mark Pagel
12	Avneesh Pant	Steve Poole	Howard Pritchard
13	Craig Rasmussen	Hubert Ritzdorf	Rob Ross
14	Tony Skjellum	Brian Smith	Vinod Tipparaju
15	Jesper Larsson Träff	Keith Underwood	
16	I		
17	The MPI Forum also ack	nowledges and appreciates the	valuable input from people via
18	e-mail and in person.		
19		s supported the $MPI-2$ effort t	hrough time and travel support
20	for the people listed above.		
21	Argonne National Labora	atory	
22	Bull		
23	Cisco Systems, Inc.		
24	Cray Inc.		
25	The HDF Group		
26	Hewlett-Packard		
27	IBM T.J. Watson Resear	ch	
28	Indiana University		
29		nerche en Informatique et Aut	omatique (Inria)
30	Intel Corporation		
31	Lawrence Berkeley Natio		
32	Lawrence Livermore Nat		
33 34	Los Alamos National Lab	poratory	
	Mathworks		
35 36	Mellanox Technologies		
37	Microsoft		
38	Myricom	NEC Emer Ltd	
39	NEC Laboratories Europ	, -	
40	Oak Ridge National Labo		
41	The Ohio State Universit		
42	Pacific Northwest Nation	al Laboratory	
43	QLogic Corporation Sandia National Laborat	aniag	
44	SiCortex	ones	
45	Silicon Graphics Incorpor	rated	
46	Sun Microsystems, Inc.	laucu	
47	University of Alabama at	Birmingham	
48	University of Houston	Diffinition	
	Oniversity of Houston		

University of Illinois at Urbana-Champaign University of Stuttgart, High Performance Computing Center Stuttgart (HLRS) University of Tennessee, Knoxville University of Wisconsin	1 2 3 4
Funding for the MPI Forum meetings was partially supported by award #CCF-0816909 from the National Science Foundation. In addition, the HDF Group provided travel support for one U.S. academic.	5 6 7 8
MPI-2.2:	9 10
All chapters have been revisited to achieve a consistent MPI-2.2 text. Those who served as authors for the necessary modifications are:	11 12 13
• William Gropp, Front matter, Introduction, and Bibliography; MPI-2.2 chair.	14
• Richard Graham, Point-to-Point Communication and Datatypes	15 16
Adam Moody, Collective Communication	17
• Torsten Hoefler, Collective Communication and Process Topologies	18 19
 Richard Treumann, Groups, Contexts, and Communicators 	20 21
	21
• Jesper Larsson Träff, Process Topologies, Info-Object and One-Sided Communications	23 24
• George Bosilca, Datatypes and Environmental Management	24 25
• David Solt, Process Creation and Management	26
• Bronis R. de Supinski, External Interfaces, and Profiling	27 28
• Rajeev Thakur, I/O	29
• Jeffrey M. Squyres, Language Bindings and MPI-2.2 Secretary	30 31
	32
• Rolf Rabenseifner, Deprecated Functions, Annex Change-Log, and Annex Language Bindings	33
	34
• Alexander Supalov, Annex Language Bindings	35 36
The following list includes some of the active participants who attended MPI-2 Forum	37
meetings and in the e-mail discussions of the errata items and are not mentioned above.	38
	39
	40
	41 42
	42
	44
	45
	46
	47
	48

1	Pavan Balaji	Purushotham V. Bangalore	Brian Barrett
2	Richard Barrett	Christian Bell	Robert Blackmore
3	Gil Bloch	Ron Brightwell	Greg Bronevetsky
4	Jeff Brown	Darius Buntinas	Jonathan Carter
5	Nathan DeBardeleben	Terry Dontje	Gabor Dozsa
6	Edric Ellis	Karl Feind	Edgar Gabriel
7	Patrick Geoffray	Johann George	David Gingold
8	David Goodell	Erez Haba	Robert Harrison
9	Thomas Herault	Marc-André Hermanns	Steve Hodson
10	Joshua Hursey	Yutaka Ishikawa	Bin Jia
11	Hideyuki Jitsumoto	Terry Jones	Yann Kalemkarian
12	Ranier Keller	Matthew Koop	Quincey Koziol
13	Manojkumar Krishnan	Sameer Kumar	Miron Livny
14	Andrew Lumsdaine	Miao Luo	Ewing Lusk
14	Timothy I. Mattox	Kannan Narasimhan	Mark Pagel
16	Avneesh Pant	Steve Poole	Howard Pritchard
17	Craig Rasmussen	Hubert Ritzdorf	Rob Ross
18	Martin Schulz	Pavel Shamis	Galen Shipman
19	Christian Siebert	Anthony Skjellum	Brian Smith
20	Naoki Sueyasu	Vinod Tipparaju	Keith Underwood
20	Rolf Vandevaart	Abhinav Vishnu	Weikuan Yu
21	Ron Vandevaart	Adminav Visiniu	Weikuan Tu
23	The MPI Forum also acknow	vledges and appreciates the va	luable input from people via
24	e-mail and in person.		
25	The following institutions su	pported the MPI-2.2 effort three	ough time and travel support
26	for the people listed above.		
27	Argonne National Laborato	rv	
28	Auburn University		
29	Bull		
30	Cisco Systems, Inc.		
31	Cray Inc.		
32	Forschungszentrum Jülich		
33	Fujitsu		
34	The HDF Group		
35	Hewlett-Packard	v	
36	International Business Mach	nines	
37	Indiana University		
38	Institut National de Rechero	che en Informatique et Autom	atique (Inria)
39		nce & Engineering Corporation	
40	Intel Corporation		
41	Lawrence Berkeley National	Laboratory	
42	Lawrence Livermore Nation		
43	Los Alamos National Labor	Ū.	
44	Mathworks	-	
45	Mellanox Technologies		
46	Microsoft		
47	Myricom		
48	NEC Corporation		

Oak Ridge National Laboratory The Ohio State University Pacific Northwest National Laboratory QLogic Corporation RunTime Computing Solutions, LLC Sandia National Laboratories SiCortex, Inc. Silicon Graphics Inc. Sun Microsystems, Inc. Tokyo Institute of Technology University of Alabama at Birmingham University of Houston University of Houston University of Stuttgart, High Performance Computing Center Stuttgart (HLRS) University of Tennessee, Knoxville University of Tokyo University of Wisconsin Funding for the MPI Forum meetings was partially supported by awards #CCF-0816909	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
and $\#$ CCF-1144042 from the National Science Foundation. In addition, the HDF Group provided travel support for one U.S. academic.	19 20 21 22
MPI-3.0: MPI-3.0 is a significant effort to extend and modernize the MPI Standard. The editors and organizers of the MPI-3.0 have been:	23 24 25 26
• William Gropp, Steering committee, Front matter, Introduction, Groups, Contexts, and Communicators, One-Sided Communications, and Bibliography	27 28
• Richard Graham, Steering committee, Point-to-Point Communication, Meeting Convener, and MPI-3.0 chair	29 30 31
• Torsten Hoefler, Collective Communication, One-Sided Communications, and Process Topologies	32 33
• George Bosilca, Datatypes and Environmental Management	34 35
• David Solt, Process Creation and Management	36 37
• Bronis R. de Supinski, External Interfaces and Tool Support	38
• Rajeev Thakur, I/O and One-Sided Communications	39 40
• Darius Buntinas, Info Object	41
• Jeffrey M. Squyres, Language Bindings and MPI-3.0 Secretary	42 43
 Rolf Rabenseifner, Steering committee, Terms and Definitions, and Fortran Bindings, Deprecated Functions, Annex Change-Log, and Annex Language Bindings 	44 45 46
• Craig Rasmussen, Fortran Bindings	47 48

Unofficial Draft for Comment Only

The following list includes some of the active participants who attended MPI-3 Forum
 meetings or participated in the e-mail discussions and who are not mentioned above.

3			
4	Tatsuya Abe	Tomoya Adachi	Sadaf Alam
5	Reinhold Bader	Pavan Balaji	Purushotham V. Bangalore
6	Brian Barrett	Richard Barrett	Robert Blackmore
7	Aurelien Bouteiller	Ron Brightwell	Greg Bronevetsky
8	Jed Brown	Darius Buntinas	Devendar Bureddy
9	Arno Candel	George Carr	Mohamad Chaarawi
10	Raghunath Raja Chandrasekar	James Dinan	Terry Dontje
11	Edgar Gabriel	Balazs Gerofi	Brice Goglin
12	David Goodell	Manjunath Gorentla	Erez Haba
13	Jeff Hammond	Thomas Herault	Marc-André Hermanns
14	Jennifer Herrett-Skjellum	Nathan Hjelm	Atsushi Hori
15	Joshua Hursey	Marty Itzkowitz	Yutaka Ishikawa
16	Nysal Jan	Bin Jia	Hideyuki Jitsumoto
17	Yann Kalemkarian	Krishna Kandalla	Takahiro Kawashima
18	Chulho Kim	Dries Kimpe	Christof Klausecker
19	Alice Koniges	Quincey Koziol	Dieter Kranzlmueller
20	Manojkumar Krishnan	Sameer Kumar	Eric Lantz
21	Jay Lofstead	Bill Long	Andrew Lumsdaine
22	Miao Luo	Ewing Lusk	Adam Moody
23	Nick M. Maclaren	Amith Mamidala	Guillaume Mercier
24	Scott McMillan	Douglas Miller	Kathryn Mohror
25	Tim Murray	Tomotake Nakamura	Takeshi Nanri
26	Steve Oyanagi	Mark Pagel	Swann Perarnau
27	Sreeram Potluri	Howard Pritchard	Rolf Riesen
28	Hubert Ritzdorf	Kuninobu Sasaki	Timo Schneider
29	Martin Schulz	Gilad Shainer	Christian Siebert
30	Anthony Skjellum	Brian Smith	Marc Snir
31	Raffaele Giuseppe Solca	Shinji Sumimoto	Alexander Supalov
32	Sayantan Sur	Masamichi Takagi	Fabian Tillier
33	Vinod Tipparaju	Jesper Larsson Träff	Richard Treumann
34	Keith Underwood	Rolf Vandevaart	Anh Vo
35	Abhinav Vishnu	Min Xie	Enqiang Zhou
36			
37	The MPI Forum also acknowled	dges and appreciates the	e valuable input from people via

The MPI Forum also acknowledges and appreciates the valuable input from people via e-mail and in person.

The MPI Forum also thanks those that provided feedback during the public comment period. In particular, the Forum would like to thank Jeremiah Wilcock for providing detailed comments on the entire draft standard.

The following institutions supported the MPI-3 effort through time and travel support for the people listed above.

Argonne National Laboratory

⁴⁵ Bull

38

44

- ⁴⁶ Cisco Systems, Inc.
- ⁴⁷ Cray Inc.
- 48 CSCS

ETH Zurich	1
Fujitsu Ltd.	2
German Research School for Simulation Sciences	3
The HDF Group	4
Hewlett-Packard	5
International Business Machines	6
IBM India Private Ltd	7
Indiana University	8
Institut National de Recherche en Informatique et Automatique (Inria)	9
Institute for Advanced Science & Engineering Corporation	10
Intel Corporation	11
Lawrence Berkeley National Laboratory	12
Lawrence Livermore National Laboratory	13
Los Alamos National Laboratory	14
Mellanox Technologies, Inc.	15
Microsoft Corporation	16
NEC Corporation	17
National Oceanic and Atmospheric Administration, Global Systems Division	18
NVIDIA Corporation	19
Oak Ridge National Laboratory	20
The Ohio State University	21
Oracle America	22
Platform Computing	23
RIKEN AICS	24
RunTime Computing Solutions, LLC	25
Sandia National Laboratories	26
Technical University of Chemnitz	27
Tokyo Institute of Technology	28
University of Alabama at Birmingham	29
University of Chicago	30
University of Houston	31
University of Illinois at Urbana-Champaign	32
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)	33
University of Tennessee, Knoxville	34
University of Tokyo	35
Funding for the MPI Forum meetings was partially supported by awards $\#$ CCF-0816909	36
and $\#CCF-1144042$ from the National Science Foundation. In addition, the HDF Group	37
and $\frac{1}{\pi}$ correct fitting hold the reaction sectore roundation. In addition, the first order and Sandia National Laboratories provided travel support for one U.S. academic each.	38
and Sandia Rational Laboratories provided traver support for one e.s. academic cach.	39
MPI-3.1:	40
IVIF I-3.1.	41
MPI-3.1 is a minor update to the MPI Standard.	42
The editors and organizers of the MPI-3.1 have been:	43
	44
• Martin Schulz, MPI-3.1 chair	45
• William Gropp, Steering committee, Front matter, Introduction, One-Sided Commu-	46
nications, and Bibliography; Overall editor	47
	48

Unofficial Draft for Comment Only

1 2	,	0	nd Definitions, and Fortran Bindings, Annex Language Bindings
3 4	• Richard L. Graham, Stee	ering committee, Meetir	ng Convener
5 6	• Jeffrey M. Squyres, Lang	guage Bindings and MP	I-3.1 Secretary
7	• Daniel Holmes, Point-to-	Point Communication	
8 9	• George Bosilca, Datatype	es and Environmental M	Management
10	• Torsten Hoefler, Collectiv	ve Communication and	Process Topologies
11 12	• Pavan Balaji, Groups, C	ontexts, and Communic	cators, and External Interfaces
13 14	• Jeff Hammond, The Info	Object	
15	• David Solt, Process Crea	·	
16 17	• Quincey Koziol, I/O	J	
18 19	• Kathryn Mohror, Tool S	upport	
20	• Rajeev Thakur, One-Side		
21 22 23 24	The following list include meetings or participated in the		articipants who attended MPI Forum
25	Charles Archer	Pavan Balaji	Purushotham V. Bangalore
26	Brian Barrett	Wesley Bland	Michael Blocksome
27	George Bosilca	Aurelien Bouteiller	Devendar Bureddy
28	Yohann Burette	Mohamad Chaarawi	Alexey Cheptsov
29	James Dinan	Dmitry Durnov	Thomas Francois
30	Edgar Gabriel	Todd Gamblin	Balazs Gerofi
31	Paddy Gillies	David Goodell	Manjunath Gorentla Venkata
32	Richard L. Graham	Ryan E. Grant	William Gropp
33	Khaled Hamidouche	Jeff Hammond	Amin Hassani
34	Marc-André Hermanns	Nathan Hjelm	Torsten Hoefler
35	Daniel Holmes	Atsushi Hori	Yutaka Ishikawa
36	Hideyuki Jitsumoto	Jithin Jose	Krishna Kandalla
37	Christos Kavouklis	Takahiro Kawashima	Chulho Kim
38	Michael Knobloch	Alice Koniges	Quincey Koziol
39	Sameer Kumar	Joshua Ladd	Ignacio Laguna
40	Huiwei Lu	Guillaume Mercier	Kathryn Mohror
41	Adam Moody	Tomotake Nakamura	Takeshi Nanri
42	Steve Oyanagi	Antonio J. Pẽna	Sreeram Potluri
43	Howard Pritchard	Rolf Rabenseifner	Nicholas Radcliffe
44	Ken Raffenetti	Raghunath Raja	Craig Rasmussen
45	Davide Rossetti	Kento Sato	Martin Schulz
46	Sangmin Seo	Christian Siebert	Anthony Skjellum
47			
- I	Brian Smith	David Solt	Jeffrey M. Squyres

	Hari Subramoni	Shinji Sumimoto	Alexander Supalov	1
	Bronis R. de Supinski	Sayantan Sur	Masamichi Takagi	2
	Keita Teranishi	Rajeev Thakur	Fabian Tillier	3
	Yuichi Tsujita	Geoffroy Vallée	Rolf vandeVaart	4
	Akshay Venkatesh	Jerome Vienne	Venkat Vishwanath	5
	Anh Vo	Huseyin S. Yildiz	Junchao Zhang	6
	Xin Zhao	Huseyin 5. Thuiz	Julienao Zhang	7
				8
The MF	PI Forum also acknowled	ges and appreciates	the valuable input from people via	9
e-mail and i	n person.			10
The foll	owing institutions suppo	rted the $MPI-3.1$ effe	rt through time and travel support	11
for the peop	le listed above.			12
Argonn	e National Laboratory			13
-	University			14
	ystems, Inc.			15
Cray	ystems, me.			16
v	The University of Edinb	urgh		17
ETH Z	-	Jurgin		18
	ngszentrum Jülich			19
Fujitsu	ingozonici ani o anon			20
•	n Research School for Sir	nulation Sciences		21
	OF Group			22
	tional Business Machines	3		23
Institut	National de Recherche	en Informatique et A	Automatique (Inria)	24
	orporation			25
	-	e, High-Performance	Computing (JARA-HPC)	26
	University	, .		27
Lawren	ce Berkeley National La	boratory		28
Lawren	ce Livermore National L	aboratory		29
Lenovo				30
Los Ala	mos National Laborator	у		31
Melland	ox Technologies, Inc.			32
Microso	oft Corporation			33
NEC C	orporation			34
NVIDL	A Corporation			35
Oak Ri	dge National Laboratory			36
	io State University			37
RIKEN				38
	National Laboratories			39
	Advanced Computing Ce	nter		40
-	Institute of Technology			41
	ity of Alabama at Birmi	ngham		42
	ity of Houston			43
	ity of Illinois at Urbana-	Champaign		44
	ity of Oregon	a		45
			ng Center Stuttgart (HLRS)	46
	ity of Tennessee, Knoxvi	ille		47
Univers	ity of Tokyo			48

Ν	/IPI-4.0:
	MPI-4.0 is a major update to the MPI Standard. The editors and organizers of the MPI-4.0 have been:
	• Martin Schulz, MPI-4.0 chair, Info Object
	• Wesley Bland, MPI-4.0 Secretary
	• William Gropp, Steering committee, Front matter, Introduction, One-Sided Commu- nications, and Bibliography; Overall editor
	• Rolf Rabenseifner, Steering committee, Terms and Conventions, Deprecated Functions, Removed Interfaces, Annex Change-Log, and Annex Language Bindings
	• Daniel Holmes, Point-to-Point Communication, Sessions
	• George Bosilca, Datatypes and Environmental Management
	• Torsten Hoefler, Collective Communication and Process Topologies
	• Pavan Balaji, Groups, Contexts, and Communicators, and External Interfaces
	• Howard Pritchard, Process Creation and Management
	• Anthony Skjellum, I/O
	• Kathryn Mohror, Tools
	• Puri Bangalore, Language Bindings
S	as part of the development of MPI-4.0, a number of working groups were established. In time cases, the work for these groups overlapped with multiple chapters. The following escribes the major working groups and the leaders of those groups:
C	Collective Communication, Topology, Communicators Torsten Hoefler, Andrew Lumsdaine, and Anthony Skjellum
F	ault Tolerance Wesley Bland, Aurélien Bouteiller, and Richard Graham
F	Iardware-Topologies Guillaume Mercier
F	Iybrid Pavan Balaji and Jim Dinan
L	arge Counts Jeff Hammond
F	Persistence Anthony Skjellum
F	Point to Point Communication Richard Graham and Dan Holmes
F	Remote Memory Access William Gropp and Rajeev Thakur
S	emantic Terms Rolf Rabenseifner and Purushotham Bangalore
S	essions Daniel Holmes
ſ	Cools Kathryn Mohror and Marc-André Hermanns

The following list includes some of the active participants who attended MPI Forum meetings or participated in the e-mail discussions.

Julien Adam Ammar Ahmad Awan Purushotham Bangalore Claudia Blaas-Schenner George Bosilca Alexander Calvert Steffen Christgau James Clark Brandon Cook Hoang-Vu Dang Murali Emani Ana Gainaru Salvatore Di Girolamo Ryan Grant Siegmar Gross Khaled Hamidouche Nathan Hjelm Atsushi Hori Julien Jaeger Jithin Jose Chulho Kim Sameer Kumar Stefan Lankes Kavitha Madhu William Marts Takeshi Nanri Lena Oden Ivy Peng Simon Pickartz Howard Pritchard **Rolf Rabenseifner** Craig Rasmussen Sergio Rivas-Gomez Amit Ruhela Sangmin Seo Min Si David Solt Hari Subramoni Savantan Sur Rajeev Thakur Geoffrov Vallee

Abdelhalim Amer Pavan Balaji Mohammadreza Bayatpour Wesley Bland Aurelien Bouteiller Nicholas Chaimov Ching-Hsiang Chu Carsten Clauss James Custer James Dinan Christian Engelmann Esthela Gallardo Brice Goglin Stanley Graves Taylor Groves Jeff Hammond Torsten Hoefler Josh Hursey **Emmanuel Jeannot** Krishna Kandalla Michael Knobloch Kim Kyunghun Tonglin Li Alexey Malhanov Guillaume Mercier Thomas Naughton Steve Oyanagi Ignacio Laguna Peralta Artem Polyakov Martina Prugger Nicholas Radcliffe Soren Rasmussen Davide Rossetti Joseph Schuchart Sameh Sharkawi Anthony Skjellum Jeff Squyres Nawrin Sultana Hugo Taboada Keith Underwood Manjunath Gorentla Venkata

Charles Archer 5Marc Gamell Balmana 6 Jean-Baptiste Besnard 7 Gil Bloch 8 Ben Bratu 9 Sourav Chakraborty 10 Mikhail Chuvelev 11 Giuseppe Congiu 12Anna Daly 13 Matthew Dosanjh 14Noah Evans 15Balazs Gerofi 16Richard Graham 17William Gropp 18 Yanfei Guo 19 Marc-André Hermanns 20Daniel Holmes 21Ilva Ivanov 22 Sylvain Jeaugev 23Takahiro Kawashima 24 Alice Koniges 25Ignacio Laguna 26Xioyi Lu 27Ryan Marshall 28Kathryn Mohror 29Takafumi Nose 30 Guillaume Papauré 31 Antonio Peña 32 Sreeram Potluri 33 Marc Pérache 34 Ken Raffenetti 35Hubert Ritzdorf 36 Martin Ruefenacht 37 Martin Schulz 38 Sameer Shende 39 Brian Smith 40 Srinivas Sridharan 41 Shinji Sumimoto 42Keita Teranishi 43 Isaias Alberto Compres Urena 44 Akshav Venkatesh 45

> 46 47 48

2

1	Jerome Vienne Anh Vo Justin Wozniak
2	Junchao Zhang Dong Zhong Hui Zhou
3	
4	
5	The MPI Ferry also admovided and appreciates the valuable input from people via
6	The MPI Forum also acknowledges and appreciates the valuable input from people via
7	e-mail and in person.
8	The following institutions supported the MPI-4.0 effort through time and travel support for the people listed above.
9	tor the people listed above.
10	ATOS/BULL
11	Argonne National Laboratory
12	Arm
13	Auburn University
14	Barcelona Supercomputing Center
15	CEA
16	Cisco Systems Inc.
17 18	Cray Inc.
19	EPCC, The University Of Edinburgh
20	ETH Zürich
20	Forschungszentrum Jülich
22	Fujitsu Fulda University of Applied Sciences
23	German Research School for Simulation Sciences
24	Hewlett Packard Enterprise
25	International Business Machines
26	Institut National de Recherche en Informatique et Automatique (Inria)
27	Intel Corporation
28	Jülich Aachen Research Alliance, High-Performance Computing (JARA-HPC)
29	Jülich Supercomputing Center
30	KTH Royal Institute Of Technology
31	Kyushu University
32	Lawrence Berkeley National Laboratory
33	Lawrence Livermore National Laboratory
34	Lenovo
35	Los Alamos National Laboratory
36	Mellanox Technologies, Inc.
37	Microsoft Corporation
38	NEC Corporation
39	NVIDIA Corporation
40	Oak Ridge National Laboratory
41	PAR-TEC
42	Paratools, Inc.
43	RIKEN AICS (R-CCS as of 2017)
44	RWTH Aachen University
45 46	Rutgers University
40 47	Sandia National Laboratories
48	Silicon Graphics, Inc.
10	VSC Research Center, TU Wien

Technical University Of Munich	1
The HDF Group	2
The Ohio State University	3
Texas Advanced Computing Center	4
Tokyo Institute of Technology	5
University of Alabama at Birmingham	6
University of Houston	7
University of Illinois at Urbana-Champaign and the National Center for Supercompu	ıt- ⁸
ing Applications	9
University of Innsbruck	10
University of Oregon	11
University of Pottsdam	12
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)	13
University of Tennessee, Chattanooga	14
University of Tennessee, Knoxville	15
University of Texas at El Paso	16
University of Tokyo	17
	18
	19
	20
	21
	22
	23
	24
	25
	26
	27
	28
	29
	30
	31 32
	32
	34
	34
	36
	37
	38
	39
	40
	41
	42
	43
	44
	45
	46
	47
	48

Chapter 1

Introduction to MPI

1.1 Overview and Goals

MPI (Message-Passing Interface) is a *message-passing library interface specification*. All parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. Extensions to the "classical" message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O. MPI is a *specification*, not an implementation; there are multiple implementations of MPI. This specification is for a *library interface*; MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings which, for C and Fortran, are part of the MPI standard. The standard has been defined through an open process by a community of parallel computing vendors, computer scientists, and application developers. The next few sections provide an overview of the history of MPI's development.

The main advantages of establishing a message-passing standard are portability and ease of use. In a distributed memory communication environment in which the higher level routines and/or abstractions are built upon lower level message-passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a messagepassing standard, such as that proposed here, provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases for which they can provide hardware support, thereby enhancing scalability.

The goal of the Message-Passing Interface simply stated is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

A complete list of goals follows.

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying, allow overlap of computation and communication, and offload to communication co-processors, where available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran bindings for the interface.

Unofficial Draft for Comment Only

 24

 31

 $45 \\ 46$

- Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
- Define an interface that can be implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.
- Semantics of the interface should be language independent.
- The interface should be designed to allow for thread safety.

1.2 Background of MPI-1.0

MPI sought to make use of the most attractive features of a number of existing messagepassing systems, rather than selecting one of them and adopting it as the standard. Thus, MPI was strongly influenced by work at the IBM T. J. Watson Research Center [2, 3], Intel's NX/2 [57], Express [14], nCUBE's Vertex [53], p4 [9, 10], and PARMACS [6, 11]. Other important contributions have come from Zipcode [60, 61], Chimp [20, 21], PVM [5, 18], Chameleon [31], and PICL [26].

18 The MPI standardization effort involved about 60 people from 40 organizations mainly 19from the United States and Europe. Most of the major vendors of concurrent computers 20were involved in MPI, along with researchers from universities, government laboratories, and 21industry. The standardization process began with the Workshop on Standards for Message-22Passing in a Distributed Memory Environment, sponsored by the Center for Research on 23Parallel Computing, held April 29–30, 1992, in Williamsburg, Virginia 69. At this work- 24 shop the basic features essential to a standard message-passing interface were discussed, 25and a working group established to continue the standardization process. 26

A preliminary draft proposal, known as MPI-1, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [19]. MPI-1 embodied the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. Since MPI-1 was primarily intended to promote discussion and "get the ball rolling," it focused mainly on point-to-point communications. MPI-1 brought to the forefront a number of important standardization issues, but did not include any collective communication routines and was not thread-safe.

In November 1992, a meeting of the MPI working group was held in Minneapolis, at 34which it was decided to place the standardization process on a more formal footing, and to 35 generally adopt the procedures and organization of the High Performance Fortran Forum. 36 Subcommittees were formed for the major component areas of the standard, and an email 37 discussion service established for each. In addition, the goal of producing a draft MPI 38 standard by the Fall of 1993 was set. To achieve this goal the MPI working group met every 39 6 weeks for two days throughout the first 9 months of 1993, and presented the draft MPI 40 standard at the Supercomputing 93 conference in November 1993. These meetings and the 41 email discussion together constituted the MPI Forum, membership of which has been open 42to all members of the high performance computing community. 43

43

45

1.3 Background of MPI-1.1, MPI-1.2, and MPI-2.0

Beginning in March 1995, the MPI Forum began meeting to consider corrections and extensions to the original MPI Standard document [23]. The first product of these deliberations

Unofficial Draft for Comment Only

7 8

9 10

11 12

13

14

15

16

17

was Version 1.1 of the MPI specification, released in June of 1995 [24] (see http://www.mpi-forum.org for official MPI document releases). At that time, effort focused in five areas.

- 1. Further corrections and clarifications for the MPI-1.1 document.
- 2. Additions to MPI-1.1 that do not significantly change its types of functionality (new datatype constructors, language interoperability, etc.).
- 3. Completely new types of functionality (dynamic processes, one-sided communication, parallel I/O, etc.) that are what everyone thinks of as "MPI-2 functionality."
- 4. Bindings for Fortran 90 and C++. MPI-2 specifies C++ bindings for both MPI-1 and MPI-2 functions, and extensions to the Fortran 77 binding of MPI-1 and MPI-2 to handle Fortran 90 issues.
- 5. Discussions of areas in which the MPI process and framework seem likely to be useful, but where more discussion and experience are needed before standardization (e.g., zero-copy semantics on shared-memory machines, real-time specifications).

Corrections and clarifications (items of type 1 in the above list) were collected in Chapter 3 of the MPI-2 document: "Version 1.2 of MPI." That chapter also contains the function for identifying the version number. Additions to MPI-1.1 (items of types 2, 3, and 4 in the above list) are in the remaining chapters of the MPI-2 document, and constitute the specification for MPI-2. Items of type 5 in the above list have been moved to a separate document, the "MPI Journal of Development" (JOD), and are not part of the MPI-2 Standard.

This structure makes it easy for users and implementors to understand what level of MPI compliance a given implementation has:

- MPI-1 compliance will mean compliance with MPI-1.3. This is a useful level of compliance. It means that the implementation conforms to the clarifications of MPI-1.1 function behavior given in Chapter 3 of the MPI-2 document. Some implementations may require changes to be MPI-1 compliant.
- MPI-2 compliance will mean compliance with all of MPI-2.1.
- The MPI Journal of Development is not part of the MPI Standard.

It is to be emphasized that forward compatibility is preserved. That is, a valid MPI-1.1 program is both a valid MPI-1.3 program and a valid MPI-2.1 program, and a valid MPI-1.3 program is a valid MPI-2.1 program.

1.4 Background of MPI-1.3 and MPI-2.1

After the release of MPI-2.0, the MPI Forum kept working on errata and clarifications for42both standard documents (MPI-1.1 and MPI-2.0). The short document "Errata for MPI-1.1"43was released October 12, 1998. On July 5, 2001, a first ballot of errata and clarifications for44MPI-2.0 was released, and a second ballot was voted on May 22, 2002. Both votes were done45electronically. Both ballots were combined into one document: "Errata for MPI-2," May4615, 2002. This errata process was then interrupted, but the Forum and its e-mail reflectors47kept working on new requests for clarification.48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5 6

7

8 9

10 11

12

13

14 15

16

17

18

19

20

21

22

23

 24

25 26

27

28

29

30

31 32

33

34 35

36

37

38 39

40

Restarting regular work of the MPI Forum was initiated in three meetings, at Eu- $\mathbf{2}$ roPVM/MPI'06 in Bonn, at EuroPVM/MPI'07 in Paris, and at SC'07 in Reno. In De-3 cember 2007, a steering committee started the organization of new MPI Forum meetings at 4 regular 8-weeks intervals. At the January 14–16, 2008 meeting in Chicago, the MPI Forum 5decided to combine the existing and future MPI documents to one document for each ver-6 sion of the MPI standard. For technical and historical reasons, this series was started with $\overline{7}$ MPI-1.3. Additional Ballots 3 and 4 solved old questions from the errata list started in 1995 8 up to new questions from the last years. After all documents (MPI-1.1, MPI-2, Errata for 9 MPI-1.1 (Oct. 12, 1998), and MPI-2.1 Ballots 1–4) were combined into one draft document, 10 for each chapter, a chapter author and review team were defined. They cleaned up the 11document to achieve a consistent MPI-2.1 document. The final MPI-2.1 standard document 12was finished in June 2008, and finally released with a second vote in September 2008 in 13 the meeting at Dublin, just before EuroPVM/MPI'08. The major work of the current MPI 14Forum is the preparation of MPI-3.

1516

1718

19

20

2122

23

 24 25

26

2728

29

30 31

32 33

34

35

36

37

Background of MPI-2.2 1.5

MPI-2.2 is a minor update to the MPI-2.1 standard. This version addresses additional errors and ambiguities that were not corrected in the MPI-2.1 standard as well as a small number of extensions to MPI-2.1 that met the following criteria:

- Any correct MPI-2.1 program is a correct MPI-2.2 program.
- Any extension must have significant benefit for users.
- Any extension must not require significant implementation effort. To that end, all such changes are accompanied by an open source implementation.

The discussions of MPI-2.2 proceeded concurrently with the MPI-3 discussions; in some cases, extensions were proposed for MPI-2.2 but were later moved to MPI-3.

1.6Background of MPI-3.0

MPI-3.0 is a major update to the MPI standard. The updates include the extension of collective operations to include nonblocking versions, extensions to the one-sided operations, and a new Fortran 2008 binding. In addition, the deprecated C++ bindings have been removed, as well as many of the deprecated routines and MPI objects (such as the MPI_UB datatype).

38 39 40

41

Background of MPI-3.1 1.7

MPI-3.1 is a minor update to the MPI standard. Most of the updates are corrections 42and clarifications to the standard, especially for the Fortran bindings. New functions added 43 include routines to manipulate MPI_Aint values in a portable manner, nonblocking collective 44 I/O routines, and routines to get the index value by name for MPI_T performance and 45control variables. A general index was also added. 46

47

48

1.8 Background of 2019 Draft Specification

The 2019 draft specification is expected to become the MPI-4.0 specification once all features have been merged. MPI-4.0 is a major update to the MPI standard. This update includes a number of new features which will be present in the final MPI-4.0 document. The largest changes are the addition of persistent collectives, application info assertions, and improvements to the definitions of error handling. In addition, there are a number of smaller improvements and corrections.

1.9 Who Should Use This Standard?

This standard is intended for use by all those who want to write portable message-passing programs in Fortran and C (and access the C bindings from C++). This includes individual application programmers, developers of software designed to run on parallel machines, and creators of environments and tools. In order to be attractive to this wide audience, the standard must provide a simple, easy-to-use interface for the basic user while not semantically precluding the high-performance message-passing operations available on advanced machines.

1.10 What Platforms Are Targets for Implementation?

The attractiveness of the message-passing paradigm at least partially stems from its wide portability. Programs expressed this way may run on distributed-memory multiprocessors, networks of workstations, and combinations of all of these. In addition, shared-memory implementations, including those for multi-core processors and hybrid architectures, are possible. The paradigm will not be made obsolete by architectures combining the sharedand distributed-memory views, or by increases in network speeds. It thus should be both possible and useful to implement this standard on a great variety of machines, including those "machines" consisting of collections of other machines, parallel or not, connected by a communication network.

The interface is suitable for use by fully general MIMD programs, as well as those written in the more restricted style of SPMD. MPI provides many features intended to improve performance on scalable parallel computers with specialized interprocessor communication hardware. Thus, we expect that native, high-performance implementations of MPI will be provided on such machines. At the same time, implementations of MPI on top of standard Unix interprocessor communication protocols will provide portability to workstation clusters and heterogenous networks of workstations.

1.11 What Is Included in the Standard?

The standard includes:

- Point-to-point communication,
- Datatypes,
- Collective operations,

6 7 8

9 10

11 12

13

14

15

16

17

2

3

4

 $\mathbf{5}$

18 19 20

21 22

23

 24

25

26

27

28

29

30

31

32

33

34

35

36

37

38 39

40 41

42 43

44 45

46 47

1	• Process groups,
2 3	• Communication contexts,
4	• Process topologies,
5 6	• Environmental management and inquiry,
7 8	• The Info object,
9	• Process creation and management,
10 11	• One-sided communication,
12 13	• External interfaces,
14	• Parallel file I/O,
15 16	• Language bindings for Fortran and C,
17 18	• Tool support.
19	
20	1.12 What Is Not Included in the Standard?
21 22	The standard does not specify:
23 24 25	• Operations that require more operating system support than is currently standard; for example, interrupt-driven receives, remote execution, or active messages,
26	• Program construction tools,
27 28	• Debugging facilities.
29 30 31 32 33 34 35	There are many features that have been considered and not included in this standard. This happened for a number of reasons, one of which is the time constraint that was self- imposed in finishing the standard. Features that are not included can always be offered as extensions by specific implementations. Perhaps future versions of MPI will address some of these issues.
36	1.13 Organization of This Document
37 38 39	The following is a list of the remaining chapters in this document, along with a brief description of each.
40 41 42	• Chapter 2, MPI Terms and Conventions, explains notational terms and conventions used throughout the MPI document.
43 44 45 46	• Chapter 3, Point-to-Point Communication, defines the basic, pairwise communication subset of MPI. <i>Send</i> and <i>receive</i> are found here, along with many associated functions designed to make basic communication powerful and efficient.
47 48	• Chapter 5, Datatypes, defines a method to describe any data layout, e.g., an array of structures in the memory, which can be used as message send or receive buffer.

Unofficial Draft for Comment Only

- Chapter 6, Collective Communication, defines process-group collective communication operations. Well known examples of this are barrier and broadcast over a group of processes (not necessarily all the processes). With MPI-2, the semantics of collective communication was extended to include intercommunicators. It also adds two new collective operations. MPI-3 adds nonblocking collective operations.
- Chapter 7, Groups, Contexts, Communicators, and Caching, shows how groups of processes are formed and manipulated, how unique communication contexts are obtained, and how the two are bound together into a *communicator*.
- Chapter 8, Process Topologies, explains a set of utility functions meant to assist in the mapping of process groups (a linearly ordered set) to richer topological structures such as multi-dimensional grids.
- Chapter 9, MPI Environmental Management, explains how the programmer can manage and make inquiries of the current MPI environment. These functions are needed for the writing of correct, robust programs, and are especially important for the construction of highly-portable message-passing programs.
- Chapter 10, The Info Object, defines an opaque object, that is used as input in several MPI routines.
- Chapter 11, Process Initialization, Creation, and Management, defines routines that allow for creation of processes.
- Chapter 12, One-Sided Communications, defines communication routines that can be completed by a single process. These include shared-memory operations (put/get) and remote accumulate operations.
- Chapter 13, External Interfaces, defines routines designed to allow developers to layer on top of MPI. This includes generalized requests, routines that decode MPI opaque objects, and threads.
- Chapter 14, I/O, defines MPI support for parallel I/O.
- Chapter 15, Tool Support, covers interfaces that allow debuggers, performance analyzers, and other tools to obtain data about the operation of MPI processes. This chapter includes Section 15.2 (Profiling Interface), which was a chapter in previous versions of MPI.
- Chapter 16, Deprecated Interfaces, describes routines that are kept for reference. However usage of these functions is discouraged, as they may be deleted in future versions of the standard.
- Chapter 17, Removed Interfaces, describes routines and constructs that have been removed from MPI. Some of these were deprecated in MPI-2, and the MPI Forum decided to remove these from the MPI-3 standard. Others of these were deprecated in MPI-3, and the MPI Forum decided to remove these from the MPI-4 standard.
- Chapter 18, Backward Incompatibilities, describes incompatibilities with previous versions of MPI.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\mathbf{5}$

 $\overline{7}$

 $45 \\ 46$

1 • Chapter 19, Language Bindings, discusses Fortran issues, and describes language in- $\mathbf{2}$ teroperability aspects between C and Fortran. 3 The Appendices are: 4 5• Annex A, Language Bindings Summary, gives specific syntax in C and Fortran, for 6 all MPI functions, constants, and types. 7 8 • Annex B, Change-Log, summarizes some changes since the previous version of the 9 standard. 10 11 • Several Index pages show the locations of examples, constants and predefined handles, 12callback routine prototypes, and all MPI functions. 13 MPI provides various interfaces to facilitate interoperability of distinct MPI imple-14mentations. Among these are the canonical data representation for MPI I/O and for 1516MPI_PACK_EXTERNAL and MPI_UNPACK_EXTERNAL. The definition of an actual bind-17ing of these interfaces that will enable interoperability is outside the scope of this document. 18 A separate document consists of ideas that were discussed in the MPI Forum during the 19MPI-2 development and deemed to have value, but are not included in the MPI Standard. They are part of the "Journal of Development" (JOD), lest good ideas be lost and in order 2021to provide a starting point for further work. The chapters in the JOD are 22• Chapter 2, Spawning Independent Processes, includes some elements of dynamic pro-23cess management, in particular management of processes with which the spawning 24processes do not intend to communicate, that the Forum discussed at length but 2526ultimately decided not to include in the MPI Standard. 27• Chapter 3, Threads and MPI, describes some of the expected interaction between an 28MPI implementation and a thread library in a multithreaded environment. 29 30 • Chapter 4, Communicator ID, describes an approach to providing identifiers for com-31municators. 32 • Chapter 5, Miscellany, discusses Miscellaneous topics in the MPI JOD, in particu-33 lar single-copy routines for use in shared-memory environments and new datatype 34 constructors. 35 36 • Chapter 6, Toward a Full Fortran 90 Interface, describes an approach to providing a 37 more elaborate Fortran 90 interface. 38 39 • Chapter 7, Split Collective Communication, describes a specification for certain non-40 blocking collective operations. 41 • Chapter 8, Real-Time MPI, discusses MPI support for real time processing. 4243 44 4546 4748

Chapter 2

MPI Terms and Conventions

This chapter explains notational terms and conventions used throughout the MPI document, some of the choices that have been made, and the rationale behind those choices.

2.1 Document Notation

Rationale. Throughout this document, the rationale for the design choices made in the interface specification is set off in this format. Some readers may wish to skip these sections, while readers interested in interface design may want to read them carefully. (*End of rationale.*)

Advice to users. Throughout this document, material aimed at users and that illustrates usage is set off in this format. Some readers may wish to skip these sections, while readers interested in programming in MPI may want to read them carefully. (*End of advice to users.*)

Advice to implementors. Throughout this document, material that is primarily commentary to implementors is set off in this format. Some readers may wish to skip these sections, while readers interested in MPI implementations may want to read them carefully. (*End of advice to implementors.*)

2.2 Naming Conventions

In many cases MPI names for C functions are of the form MPI_Class_action_subset. This convention originated with MPI-1. Since MPI-2 an attempt has been made to standardize the names of MPI functions according to the following rules.

- 1. In C, all routines associated with a particular type of MPI object should be of the form MPI_Class_action_subset or, if no subset exists, of the form MPI_Class_action. In Fortran, all routines associated with a particular type of MPI object should be of the form MPI_CLASS_ACTION_SUBSET or, if no subset exists, of the form MPI_CLASS_ACTION.
- 2. If the routine is not associated with a class, the name should be of the form MPI_Action_subset in C and MPI_ACTION_SUBSET in Fortran.

 $45 \\ 46$

3. The names of certain actions have been standardized. In particular, **Create** creates a new object, Get retrieves information about an object, Set sets this information, **Delete** deletes information, **Is** asks whether or not an object has a certain property.

C and Fortran names for some MPI functions (that were defined during the MPI-1 process) violate these rules in several cases. The most common exceptions are the omission of the Class name from the routine and the omission of the Action where one can be inferred.

MPI identifiers are limited to 30 characters (31 with the profiling interface). This is done to avoid exceeding the limit on some compilation systems.

Procedure Specification 2.3

MPI procedures are specified using a language-independent notation. The arguments of procedure calls are marked as IN, OUT, or INOUT. The meanings of these are:

• IN: the call may use the input value but does not update the argument from the perspective of the caller at any time during the call's execution,

• OUT: the call may update the argument but does not use its input value,

• INOUT: the call may both use and update the argument.

There is one special case — if an argument is a handle to an opaque object (these 23terms are defined in Section 2.5.1), and the object is updated by the procedure call, then 24 the argument is marked INOUT or OUT. It is marked this way even though the handle itself 2526is not modified — we use the INOUT or OUT attribute to denote that what the handle references is updated. 27

28 29

30

3132

34

37

Rationale. The definition of MPI tries to avoid, to the largest possible extent, the use of INOUT arguments, because such use is error-prone, especially for scalar arguments. (End of rationale.)

MPI's use of IN. OUT, and INOUT is intended to indicate to the user how an argument 33 is to be used, but does not provide a rigorous classification that can be translated directly into all language bindings (e.g., INTENT in Fortran 90 bindings or const in C bindings). 35 For instance, the "constant" MPI_BOTTOM can usually be passed to OUT buffer arguments. 36 Similarly, MPI_STATUS_IGNORE can be passed as the OUT status argument.

A common occurrence for MPI functions is an argument that is used as IN by some pro-38 cesses and OUT by other processes. Such an argument is, syntactically, an INOUT argument 39 and is marked as such, although, semantically, it is not used in one call both for input and 40 for output on a single process. 41

Another frequent situation arises when an argument value is needed only by a subset 42of the processes. When an argument is not significant at a process then an arbitrary value 43 can be passed as an argument. 44

Unless specified otherwise, an argument of type OUT or type INOUT cannot be aliased 45with any other argument passed to an MPI procedure. An example of argument aliasing in 46 C appears below. If we define a C procedure like this, 47

48

1

2

3

4

5

6

 $\overline{7}$ 8

9

10 11 12

13 14

15

16

17

18 19

2021

```
void copyIntBuffer(int *pin, int *pout, int len)
{    int i;
    for (i=0; i<len; ++i) *pout++ = *pin++;
}</pre>
```

then a call to it in the following code fragment has aliased arguments.

int a[10]; copyIntBuffer(a, a+3, 7);

Although the C language allows this, such usage of MPI procedures is forbidden unless otherwise specified. Note that Fortran prohibits aliasing of arguments.

All MPI functions are first specified in the language-independent notation. Immediately below this, language dependent bindings follow:

- The ISO C version of the function.
- The Fortran version used with USE mpi_f08.
- The Fortran version of the same function used with USE mpi or INCLUDE 'mpif.h'.

An exception is Section 15.3 "The MPI Tool Information Interface", which only provides ISO C interfaces.

"Fortran" in this document refers to Fortran 90 and higher; see Section 2.6.

The words function, routine, procedure, procedure call, and call are often used as synonyms within this standard.

2.4 Semantic Terms

When discussing MPI procedures the following semantic terms are used. The term **message** data buffer refers to the send/receive buffer used in a communication procedure. The term file data buffer refers to the data buffers used by MPI I/O procedures. In this section we use the term data buffer and depending on the MPI procedure it will refer to message data buffer or file data buffer.

2.4.1 MPI Operations

- **MPI operation** An MPI operation is a sequence of steps performed by the MPI library to establish and enable data transfer and/or synchronization. It consists of four stages: initialization, starting, completion, and freeing, and it is implemented as a set of one or more MPI procedures, see Section 2.4.2.
 - **Initialization** hands over the argument list to the operation but not the content of the data buffers, if any. The specification of an operation may state that array arguments must not be changed until the operation is freed.
 - **Starting** hands over the control of the data buffers, if any, to the associated operation.

Note that **initiation** refers to the combination of the initialization and starting stages.

 $45 \\ 46$

	12 CHAPTER 2. MPI TERMS AND CONVENTIONS
1 2 3 4 5 6 7	 Completion returns control of the content of the data buffers to the application and indicates that output buffers and arguments, if any, have been updated. Note that an MPI operation is complete when the MPI procedure implementing the completion stage returns. Freeing returns control of the rest of the argument list (e.g., the data buffer address and array arguments).
8 9	MPI operations are available in one or more of these forms: blocking, nonblocking, and persistent.
10 11 12	Blocking operation For a blocking operation, all four stages are combined in a single procedure call (as shown in Figure 2.1 and defined in Section 2.4.2).
13 14 15 16 17	Initialization & Starting Completion & Freeing Figure 2.1: State Transition Diagram for Blocking Operations
18 19 20 21 22 23	Nonblocking operation For a nonblocking operation, the initialization and starting stages are combined into a single nonblocking procedure call and the completion and freeing stages are combined into a separate, single procedure call, which can be blocking or nonblocking (as shown in Figure 2.2 and defined in Section 2.4.2).
24 25 26 27 28 29 30 31	Figure 2.2: State Transition Diagram for Nonblocking Operations
32 33 34	Persistent operation For a persistent operation , there is a separate procedure for each of the four stages (as shown in Figure 2.3 and defined in Section 2.4.2). Each of these procedures may be blocking or nonblocking.
35 36 37 38 39 40	For a partitioned send operation, an additional call to activate each partition of the send buffer (see Section 4.2.1) is required to finish the starting stage. For a partitioned receive operation, before the operation is complete the user is allowed to access a partition of the output buffer after verifying that it has arrived (see Section ??). Additionally, an MPI operation can be collective or noncollective.
41 42 43 44 45 46	Collective operation Collective operations are defined as operations that involve a group or groups of MPI processes. For collective operations the completion stage may or may not finish before all processes in the group have started the operation.Collective MPI operations are also available as blocking, nonblocking, or persistent operations.
47 48	Noncollective operation Noncollective operations are defined as operations that are not collective.



Figure 2.3: State Transition Diagram for Persistent Operations

2.4.2 MPI Procedures

All MPI procedures can either be local or non-local - defined as follows:

Non-local procedure An MPI procedure is non-local if returning may require, during its execution, some specific semantically-related MPI procedure to be called on another MPI process.

Local procedure An MPI procedure is local if it is not non-local.

An MPI operation is implemented as a set of one or more MPI procedures. An MPI **operation-related procedure** implements at least a part of a stage of an MPI operation as described in Section 2.4.1. An MPI operation-related procedure may also implement one or more stages of one or several MPI operations. In certain cases, more than one MPI operation-related procedure may be needed to implement a single stage.

There are also other MPI procedures that do not implement any stage of any MPI operation.

The semantics of MPI operation-related procedures are described using two orthogonal (independent) concepts: completeness (depends on which stages are included) and locality. Such procedures can be either incomplete, or completing, or freeing, or completing and freeing based on the status of the associated operation at the time the procedure returns. Also, all such procedures can be described as either blocking or nonblocking, but these latter two terms refer to combinations of the completeness and locality concepts. Additionally, all MPI operation-related procedures can be collective or noncollective.

The following are properties of MPI operation-related procedures:

- **Initialization procedure** An MPI procedure is an **initialization procedure** if return from the procedure indicates that the associated operation has completed its initialization stage, which implies that the user has handed over control of the argument list (but not contents of the data buffers) to MPI. The user is still allowed to read or modify the contents of the data buffers. If an initializing procedure is not also the freeing procedure of the associated operation (see below) then the user is not permitted to deallocate the data buffers or to modify the array arguments.
- **Starting procedure** An MPI procedure is a **starting procedure** if return from the procedure indicates that the associated operation has completed its starting stage, which implies that the user has handed over control of the data buffers to MPI. If a starting procedure is not also a completing procedure of the associated operation (see below) then the user is not permitted to modify input data buffers or to read output data buffers.

 24

1 2 3	Initiation procedure An MPI procedure is an initiation procedure if return from the procedure indicates that both the initialization and the starting stage have completed, which implies control of the entire argument list is handed over to MPI.
4 5 6 7 8 9 10	Completing procedure An MPI procedure is called completing if return from the procedure indicates that at least one associated operation has finished its completion stage, which implies that the user can rely on the content of the output data buffers and modify the content of input and output data buffers of such operation(s). If a completing procedure is not also a freeing procedure (see below) then the user is not permitted to deallocate the data buffers or to modify the array arguments.
11 12 13	Incomplete procedure An MPI procedure is called incomplete if it is not a completing procedure.
14 15 16 17 18 19	 Freeing procedure An MPI procedure is freeing if return from the procedure indicates that at least one associated operation has finished its freeing stage, which implies that the user can reuse all parameters specified when initializing such associated operation(s). Nonblocking procedure An MPI procedure is nonblocking if it is incomplete and local.
20 21	Blocking procedure An MPI procedure is blocking if it is not nonblocking.
22 23 24 25 26	Advice to users. Note that for operation-related MPI procedures, in most cases incomplete procedures are local and completing procedures are non-local. Exceptions are noted where such procedures are defined. In many cases an additional prefix letter I as an abbreviation of the words incomplete and immediate marks nonblocking procedures in the procedure name.
27	Some categorization examples are listed below.
28 29	Nonblocking procedures:
30 31 32	• incomplete and local: MPI_ISEND, MPI_IRECV, MPI_IBCAST, MPI_IMPROBE, MPI_SEND_INIT, MPI_RECV_INIT,
33	Blocking procedures:
34 35	• completing and non-local: MPI_SEND, MPI_RECV, MPI_BCAST,
36 37	 incomplete and non-local: MPI_MPROBE, MPI_BCAST_INIT,, MPI_FILE_{READ WRITE}_{AT_ALL ALL ORDERED}_BEGIN.
38	• completing and local: MPI_BSEND, MPI_RSEND, MPI_MRECV.
39 40	MPI procedures that are not MPI operation-related:
41	• MPI_COMM_RANK, MPI_WTIME, MPI_PROBE, MPI_IPROBE,
42 43	(End of advice to users.)
44 45 46	Collective procedure An MPI procedure is collective if all processes in a group or groups of MPI processes need to invoke the procedure.
47 48	Initialization procedures of collective operations over the same process group must be executed in the same order by all members of the process group.

Unofficial Draft for Comment Only

An MPI collective procedure is **synchronizing** if it will only return once all processes in the associated group or groups of MPI processes have called the appropriate matching MPI procedure.

The initiation procedures for nonblocking collective operations and the starting procedures for persistent collective operations are local and shall not be synchronizing.

All other procedures for collective operations, such as for blocking collective operations and the initialization procedures for persistent collective operations, may or may not be synchronizing.

Calling any synchronizing function is erroneous when there is no Advice to users. possibility of corresponding calls at all other processes in the associated process group.

Waiting for completion of any collective operation is erroneous when there is no possibility that all other processes in the associated group will be able to start the corresponding operation. (End of advice to users.)

2.4.3 **MPI** Datatypes

For datatypes, the following terms are defined:

predefined A predefined datatype is a datatype with a predefined (constant) name (such as MPI_INT, MPI_FLOAT_INT, or MPI_PACKED) or a datatype constructed with MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL, or MPI_TYPE_CREATE_F90_COMPLEX. The former are named whereas the latter are unnamed.

derived A derived datatype is any datatype that is not predefined.

- portable A datatype is portable if it is a predefined datatype, or it is derived from 28 a portable datatype using only the type constructors MPI_TYPE_CONTIGUOUS, 29MPI_TYPE_VECTOR, MPI_TYPE_INDEXED, 30 MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_TYPE_CREATE_SUBARRAY, MPI_TYPE_DUP, and MPI_TYPE_CREATE_DARRAY. Such a datatype is portable 32 because all displacements in the datatype are in terms of extents of one predefined 33 datatype. Therefore, if such a datatype fits a data layout in one memory, it will 34 fit the corresponding data layout in another memory, if the same declarations were 35 used, even if the two systems have different architectures. On the other hand, if a 36 datatype was constructed using MPI_TYPE_CREATE_HINDEXED, 37 MPI_TYPE_CREATE_HINDEXED_BLOCK, MPI_TYPE_CREATE_HVECTOR or 38 MPI_TYPE_CREATE_STRUCT, then the datatype contains explicit byte displace-39 ments (e.g., providing padding to meet alignment restrictions). These displacements are unlikely to be chosen correctly if they fit data layout on one memory, but are 41 used for data layouts on another process, running on a processor with a different 42architecture. 44
- equivalent Two datatypes are equivalent if they appear to have been created with the same sequence of calls (and arguments) and thus have the same typemap. Two equivalent datatypes do not necessarily have the same cached attributes or the same names.

4748

40

43

45

46

1 2

3

4

5

6

 $\overline{7}$

8

9 10

11

12

13

14

151617

18

19 20

21

22

23

24

2526

27

 31

2.5 Data Types

2.5.1 Opaque Objects

MPI manages **system memory** that is used for buffering messages and for storing internal representations of various MPI objects such as groups, communicators, datatypes, etc. This memory is not directly accessible to the user, and objects stored there are **opaque**: their size and shape is not visible to the user. Opaque objects are accessed via **handles**, which exist in user space. MPI procedures that operate on opaque objects are passed handle arguments to access these objects. In addition to their use by MPI calls for object access, handles can participate in assignments and comparisons.

In Fortran with USE mpi or INCLUDE 'mpif.h', all handles have type INTEGER. In Fortran with USE mpi_f08, and in C, a different handle type is defined for each category of objects. With Fortran USE mpi_f08, the handles are defined as Fortran BIND(C) derived types that consist of only one element INTEGER :: MPI_VAL. The internal handle value is identical to the Fortran INTEGER value used in the mpi module and mpif.h. The operators .EQ., .NE., == and /= are overloaded to allow the comparison of these handles. The type names are identical to the names in C, except that they are not case sensitive. For example:

²⁰ TYPE, BIND(C) :: MPI_Comm

- INTEGER :: MPI_VAL
- ²² END TYPE MPI_Comm

The C types must support the use of the assignment and equality operators.

Advice to implementors. In Fortran, the handle can be an index into a table of opaque objects in a system table; in C it can be such an index or a pointer to the object. (End of advice to implementors.)

Rationale. Since the Fortran integer values are equivalent, applications can easily convert MPI handles between all three supported Fortran methods. For example, an integer communicator handle COMM can be converted directly into an exactly equivalent mpi_f08 communicator handle named comm_f08 by comm_f08%MPI_VAL=COMM, and vice versa. The use of the INTEGER defined handles and the BIND(C) derived type handles is different: Fortran 2003 (and later) define that BIND(C) derived types can be used within user defined common blocks, but it is up to the rules of the companion C compiler how many numerical storage units are used for these BIND(C) derived type handles. Most compilers use one unit for both, the INTEGER handles and the handles defined as BIND(C) derived types. (End of rationale.)

Advice to users. If a user wants to substitute mpif.h or the mpi module by the mpi_f08 module and the application program stores a handle in a Fortran common block then it is necessary to change the Fortran support method in all application routines that use this common block, because the number of numerical storage units of such a handle can be different in the two modules. (End of advice to users.)

⁴⁶ Opaque objects are allocated and deallocated by calls that are specific to each object
 ⁴⁷ type. These are listed in the sections where the objects are described. The calls accept a
 ⁴⁸ handle argument of matching type. In an allocate call this is an OUT argument that returns

Unofficial Draft for Comment Only

3 4

5

6

7

8

9

10

19

21

23

24 25

26

27

28 29

30

31

32

33

34

35

36

37

38

39 40

41

42

43

44

45

a valid reference to the object. In a call to deallocate this is an INOUT argument which returns with an "invalid handle" value. MPI provides an "invalid handle" constant for each object type. Comparisons to this constant are used to test for validity of the handle.

A call to a deallocate routine invalidates the handle and marks the object for deallocation. The object is not accessible to the user after the call. However, MPI need not deallocate the object immediately. Any operation pending (at the time of the deallocate) that involves this object will complete normally; the object will be deallocated afterwards.

An opaque object and its handle are significant only at the process where the object was created and cannot be transferred to another process.

MPI provides certain predefined opaque objects and predefined, static handles to these objects. The user must not free such objects.

Rationale. This design hides the internal representation used for MPI data structures, thus allowing similar calls in C and Fortran. It also avoids conflicts with the typing rules in these languages, and easily allows future extensions of functionality. The mechanism for opaque objects used here loosely follows the POSIX Fortran binding standard.

The explicit separation of handles in user space and objects in system space allows space-reclaiming and deallocation calls to be made at appropriate points in the user program. If the opaque objects were in user space, one would have to be very careful not to go out of scope before any pending operation requiring that object completed. The specified design allows an object to be marked for deallocation, the user program can then go out of scope, and the object itself still persists until any pending operations are complete.

The requirement that handles support assignment/comparison is made since such operations are common. This restricts the domain of possible implementations. The alternative in C would have been to allow handles to have been an arbitrary, opaque type. This would force the introduction of routines to do assignment and comparison, adding complexity, and was therefore ruled out. In Fortran, the handles are defined such that assignment and comparison are available through the operators of the language or overloaded versions of these operators. (*End of rationale.*)

Advice to users. A user may accidentally create a dangling reference by assigning to a handle the value of another handle, and then deallocating the object associated with these handles. Conversely, if a handle variable is deallocated before the associated object is freed, then the object becomes inaccessible (this may occur, for example, if the handle is a local variable within a subroutine, and the subroutine is exited before the associated object is deallocated). It is the user's responsibility to avoid adding or deleting references to opaque objects, except as a result of MPI calls that allocate or deallocate such objects. (*End of advice to users.*)

Advice to implementors. The intended semantics of opaque objects is that opaque objects are separate from one another; each call to allocate such an object copies all the information required for the object. Implementations may avoid excessive copying by substituting referencing for copying. For example, a derived datatype may contain references to its components, rather than copies of its components; a call to MPI_COMM_GROUP may return a reference to the group associated with the communicator, rather than a copy of this group. In such cases, the implementation 48

Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$

6 7

8 9

10

11 12

13

14

15

16

17 18

19

20

21

22

23

 24

25

26

27

28

29

30

 31

32 33

34

35

36

37

38

39

40

must maintain reference counts, and allocate and deallocate objects in such a way that the visible effect is as if the objects were copied. (End of advice to implementors.)

2.5.2 Array Arguments

An MPI call may need an argument that is an array of opaque objects, or an array of 6 handles. The array-of-handles is a regular array with entries that are handles to objects of the same type in consecutive locations in the array. Whenever such an array is used, 8 an additional len argument is required to indicate the number of valid entries (unless this 9 number can be derived otherwise). The valid entries are at the beginning of the array; 10 len indicates how many of them there are, and need not be the size of the entire array. 11 The same approach is followed for other array arguments. In some cases NULL handles are 12considered valid entries. When a NULL argument is desired for an array of statuses, one 13 uses MPI_STATUSES_IGNORE. 14

2.5.3 State

17MPI procedures use at various places arguments with state types. The values of such a data 18 type are all identified by names, and no operation is defined on them. For example, the 19MPI_TYPE_CREATE_SUBARRAY routine has a state argument order with values 20MPI_ORDER_C and MPI_ORDER_FORTRAN. 21

22 23

15

16

2.5.4 Named Constants

 24 MPI procedures sometimes assign a special meaning to a special value of a basic type argu-25ment; e.g., tag is an integer-valued argument of point-to-point communication operations, 26with a special wild-card value, MPI_ANY_TAG. Such arguments will have a range of regular 27values, which is a proper subrange of the range of values of the corresponding basic type; 28special values (such as MPI_ANY_TAG) will be outside the regular range. The range of regu-29lar values, such as tag, can be queried using environmental inquiry functions, see Chapter 9. 30 The range of other values, such as source, depends on values given by other MPI routines 31 (in the case of **source** it is the communicator size).

32 MPI also provides predefined named constant handles, such as MPI_COMM_WORLD.

33 All named constants, with the exceptions noted below for Fortran, can be used in 34 initialization expressions or assignments, but not necessarily in array declarations or as 35 labels in C switch or Fortran select/case statements. This implies named constants 36 to be link-time but not necessarily compile-time constants. The named constants listed 37 below are required to be compile-time constants in both C and Fortran. These constants 38do not change values during execution. Opaque objects accessed by constant handles are 39 defined and do not change value between MPI initialization (MPI_INIT) and MPI completion 40(MPI_FINALIZE). The handles themselves are constants and can be also used in initialization 41 expressions or assignments.

42The constants that are required to be compile-time constants (and can thus be used 43for array length declarations and labels in C switch and Fortran case/select statements)

- 44
- 45
- 4647
- 48

1

 $\mathbf{2}$

3 4

5

oro	٠
are	

MPI_MAX_PROCESSOR_NAME
MPI_MAX_LIBRARY_VERSION_STRING
MPI_MAX_ERROR_STRING
MPI_MAX_DATAREP_STRING
MPI_MAX_INFO_KEY
MPI_MAX_INFO_VAL
MPI_MAX_OBJECT_NAME
MPI_MAX_PORT_NAME
MPI_VERSION
MPI_SUBVERSION
MPI_F_STATUS_SIZE (C only)
MPI_STATUS_SIZE (Fortran only)
MPI_ADDRESS_KIND (Fortran only)
MPI_COUNT_KIND (Fortran only)
MPI_INTEGER_KIND (Fortran only)
MPI_OFFSET_KIND (Fortran only)
MPI_SUBARRAYS_SUPPORTED (Fortran only)
MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)
The constants that cannot be used in initialization expressions or assignments in For-
tran are as follows:
MPI_BOTTOM
MPI_STATUS_IGNORE
MPI_STATUSES_IGNORE
MPI_ERRCODES_IGNORE
MPI_IN_PLACE MPI_ARGV_NULL
MPI_ARGV_NULL
MPI_UNWEIGHTED
MPI_WEIGHTS_EMPTY
Advice to implementors. In Fortran the implementation of these special constants

Advice to implementors. In Fortran the implementation of these special constants may require the use of language constructs that are outside the Fortran standard. Using special values for the constants (e.g., by defining them through PARAMETER statements) is not possible because an implementation cannot distinguish these values from valid data. Typically, these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared COMMON block), relying on the fact that the target compiler passes data by address. Inside the subroutine, this address can be extracted by some mechanism outside the Fortran standard (e.g., by Fortran extensions or by implementing the function in C). (End of advice to implementors.)

2.5.5 Choice

MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types. The mechanism for providing such arguments will differ from language to language. For Fortran with the include file mpif.h or the mpi module, the document uses <type> to represent a choice variable; with the Fortran mpi_f08 module, such arguments are declared with the Fortran 2008 + TS 29113 syntax TYPE(*), DIMENSION(..); for C, we use void*.

Unofficial Draft for Comment Only

Advice to implementors. Implementors can freely choose how to implement choice arguments in the mpi module, e.g., with a non-standard compiler-dependent method that has the quality of the call mechanism in the implicit Fortran interfaces, or with the method defined for the mpi_f08 module. See details in Section 19.1.1. (End of advice to implementors.)

2.5.6 Absolute Addresses and Relative Address Displacements

Some MPI procedures use address arguments that represent an absolute address in the call ing program, or relative displacement arguments that represent differences of two absolute
 addresses. The datatype of such arguments is MPI_Aint in C and INTEGER (KIND=
 MPI_ADDRESS_KIND) in Fortran. These types must have the same width and encode address
 values in the same manner such that address values in one language may be passed directly
 to another language without conversion. There is the MPI constant MPI_BOTTOM to in-

dicate the start of the address range. For retrieving absolute addresses or any calculation with absolute addresses, one should use the routines and functions provided in Section 5.1.5. Section 5.1.12 provides additional rules for the correct use of absolute addresses. For expressions with relative displacements or other usage without absolute addresses, intrinsic operators (e.g., +, -, *) can be used.

2.5.7 File Offsets

For I/O there is a need to give the size, displacement, and offset into a file. These quantities can easily be larger than 32 bits which can be the default size of a Fortran integer. To overcome this, these quantities are declared to be INTEGER (KIND=MPI_OFFSET_KIND) in Fortran. In C one uses MPI_Offset. These types must have the same width and encode address values in the same manner such that offset values in one language may be passed directly to another language without conversion.

²⁹ 2.5.8 Counts

 31 As described above, MPI defines types (e.g., MPI_Aint) to address locations within memory 32 and other types (e.g., MPI_Offset) to address locations within files. In addition, some MPI 33 procedures use *count* arguments that represent a number of MPI datatypes on which to 34operate. Furthermore, timestamps in the context of the MPI Tool Information Interface are 35 a count of clock ticks elapsed since some time in the past. At times, one needs a single 36 type that can be used to address locations within either memory or files as well as express 37 count values, and that type is MPI_Count in C and INTEGER (KIND=MPI_COUNT_KIND) in 38Fortran. These types must have the same width and encode values in the same manner 39 such that count values in one language may be passed directly to another language without 40conversion. The size of the MPI_Count type is determined by the MPI implementation 41 with the restriction that it must be minimally capable of encoding any value that may 42be stored in a variable of type int, MPI_Aint, or MPI_Offset in C and of type INTEGER, 43INTEGER (KIND=MPI_ADDRESS_KIND), or INTEGER (KIND=MPI_OFFSET_KIND) in Fortran.

 $44 \\ 45$

46 47

48

Rationale. Count values logically need to be large enough to encode any value used for expressing element counts, type maps in memory, type maps in file views, etc. For backward compatibility reasons, many MPI routines still use int in C and INTEGER in Fortran as the type of count arguments. (*End of rationale.*)

Unofficial Draft for Comment Only

1

2

3

4

5

6 7

8

20

2.6 Language Binding

This section defines the rules for MPI language binding in general and for Fortran, and ISO C, in particular. (Note that ANSI C has been replaced by ISO C.) Defined here are various object representations, as well as the naming conventions used for expressing this standard. The actual calling sequences are defined elsewhere.

MPI bindings are for Fortran 90 or later, though they were originally designed to be usable in Fortran 77 environments. With the mpi_f08 module, two new Fortran features, assumed type and assumed rank, are also required, see Section 2.5.5.

Since the word **PARAMETER** is a keyword in the Fortran language, we use the word "argument" to denote the arguments to a subroutine. These are normally referred to as parameters in C, however, we expect that C programmers will understand the word "argument" (which has no specific meaning in C), thus allowing us to avoid unnecessary confusion for Fortran programmers.

Since Fortran is case insensitive, linkers may use either lower case or upper case when resolving Fortran names. Users of case sensitive languages should avoid any prefix of the form "MPI_" and "PMPI_", where any of the letters are either upper or lower case.

2.6.1 Deprecated and Removed Interfaces

A number of chapters refer to deprecated or replaced MPI constructs. These are constructs that continue to be part of the MPI standard, as documented in Chapter 16, but that users are recommended not to continue using, since better solutions were provided with newer versions of MPI. For example, the Fortran binding for MPI-1 functions that have address arguments uses INTEGER. This is not consistent with the C binding, and causes problems on machines with 32 bit INTEGERs and 64 bit addresses. In MPI-2, these functions were given new names with new bindings for the address arguments. The use of the old functions was declared as deprecated. For consistency, here and in a few other cases, new C functions are also provided, even though the new functions are equivalent to the old functions. The old names are deprecated.

Some of the deprecated constructs are now removed, as documented in Chapter 17. They may still be provided by an implementation for backwards compatibility, but are not required.

Table 2.1 shows a list of all of the deprecated and removed constructs. Note that some C typedefs and Fortran subroutine names are included in this list; they are the types of callback functions.

2.6.2 Fortran Binding Issues

Originally, MPI-1.1 provided bindings for Fortran 77. These bindings are retained, but they are now interpreted in the context of the Fortran 90 standard. MPI can still be used with most Fortran 77 compilers, as noted below. When the term "Fortran" is used it means Fortran 90 or later; it means Fortran 2008 + TS 29113 and later if the mpi_f08 module is used.

All MPI names have an MPI_ prefix, and all characters are capitals. Programs must ⁴⁴ not declare names, e.g., for variables, subroutines, functions, parameters, derived types, ⁴⁵ abstract interfaces, or modules, beginning with the prefix MPI_. To avoid conflicting with ⁴⁶ the profiling interface, programs must also avoid subroutines and functions with the prefix ⁴⁷ PMPI_. This is mandated to avoid possible name collisions. ⁴⁸

Unofficial Draft for Comment Only

 $\overline{7}$

 24

1	Deprecated or removed	deprecated	removed	Replacement
2	construct	since	since	•
3	MPI_ADDRESS	MPI-2.0	MPI-3.0	MPI_GET_ADDRESS
	MPI_TYPE_HINDEXED	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HINDEXED
4	MPI_TYPE_HVECTOR	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HVECTOR
5	MPI_TYPE_STRUCT	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_STRUCT
6	MPI_TYPE_EXTENT	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
7	MPI_TYPE_UB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
	MPI_TYPE_LB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
8	MPI_LB ¹	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED
9	MPI_UB ¹	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED
10	MPI_ERRHANDLER_CREATE	MPI-2.0	MPI-3.0	MPI_COMM_CREATE_ERRHANDLER
11	MPI_ERRHANDLER_GET	MPI-2.0	MPI-3.0	MPI_COMM_GET_ERRHANDLER
12	MPI_ERRHANDLER_SET	MPI-2.0	MPI-3.0	MPI_COMM_SET_ERRHANDLER
	MPI_Handler_function ²	MPI-2.0	MPI-3.0	$MPI_Comm_errhandler_function^2$
13	MPI_KEYVAL_CREATE	MPI-2.0		MPI_COMM_CREATE_KEYVAL
14	MPI_KEYVAL_FREE	MPI-2.0		MPI_COMM_FREE_KEYVAL
15	MPI_DUP_FN ³	MPI-2.0		MPI_COMM_DUP_FN ³
16	MPI_NULL_COPY_FN ³	MPI-2.0		MPI_COMM_NULL_COPY_FN ³
	MPI_NULL_DELETE_FN ³	MPI-2.0		MPI_COMM_NULL_DELETE_FN ³
17	MPI_Copy_function ²	MPI-2.0		MPI_Comm_copy_attr_function ²
18	COPY_FUNCTION ³	MPI-2.0		COMM_COPY_ATTR_FUNCTION ³
19	MPI_Delete_function ²	MPI-2.0		MPI_Comm_delete_attr_function ²
20	DELETE_FUNCTION ³	MPI-2.0		COMM_DELETE_ATTR_FUNCTION ³
21	MPI_ATTR_DELETE	MPI-2.0		MPI_COMM_DELETE_ATTR
	MPI_ATTR_GET	MPI-2.0		MPI_COMM_GET_ATTR
22	MPI_ATTR_PUT	MPI-2.0		MPI_COMM_SET_ATTR
23	MPI_COMBINER_HVECTOR_INTEGER ⁴	-	MPI-3.0	MPI_COMBINER_HVECTOR ⁴
24	MPI_COMBINER_HINDEXED_INTEGER ⁴	-	MPI-3.0	MPI_COMBINER_HINDEXED ⁴
25	MPI_COMBINER_STRUCT_INTEGER ⁴	-	MPI-3.0	MPI_COMBINER_STRUCT ⁴
26	MPI:	MPI-2.2	MPI-3.0	C language binding
	MPI_CANCEL for send requests	MPI-3.2		no direct replacement
27	MPI_INFO_GET	MPI-4.0		MPI_INFO_GET_STRING
28	MPI_INFO_GET_VALUELEN	MPI-4.0		MPI_INFO_GET_STRING
29	MPI_T_ERR_INVALID_ITEM	MPI-3.2		MPI_T_ERR_INVALID_INDEX
30	MPI_SIZEOF	MPI-4.0	•	storage_size() ⁵
31	¹ Predefined datatype.			
32	 ² Callback prototype definition. ³ Predefined callback routine. 			
	⁴ Constant.			
33	⁵ Fortran intrinsic. It returns the size in	n bits instead	of bytes	
34	Other entries are regular MPI routines.	n bitb inbtodd	or by tob.	
35				
36		-	1.5	
37	Table 2.1: De	eprecated a	nd Remov	ed constructs
38				

All MPI Fortran subroutines have a return code in the last argument. With USE mpi_f08, this last argument is declared as OPTIONAL, except for user-defined callback functions (e.g., COMM_COPY_ATTR_FUNCTION) and their predefined callbacks (e.g.,

MPI_NULL_COPY_FN). A few MPI operations which are functions do not have the return code argument. The return code value for successful completion is MPI_SUCCESS. Other error codes are implementation dependent; see the error codes in Chapter 9 and Annex A. Constants representing the maximum length of a string are one smaller in Fortran than in C as discussed in Section 19.2.9.

Handles are represented in Fortran as INTEGERs, or as a BIND(C) derived type with the

mpi_f08 module; see Section 2.5.1. Binary-valued variables are of type LOGICAL.

Array arguments are indexed from one.

The older MPI Fortran bindings (mpif.h and use mpi) are inconsistent with the Fortran standard in several respects. These inconsistencies, such as register optimization problems, have implications for user codes that are discussed in detail in Section 19.1.16.

2.6.3 C Binding Issues

We use the ISO C declaration format. All MPI names have an MPI_ prefix, defined constants are in all capital letters, and defined types and functions have one capital letter after the prefix. Programs must not declare names (identifiers), e.g., for variables, functions, constants, types, or macros, beginning with any prefix of the form MPI_, where any of the letters are either upper or lower case. To support the profiling interface, programs must not declare functions with names beginning with any prefix of the form PMPI_, where any of the letters are either upper or lower case.

The definition of named constants, function prototypes, and type definitions must be supplied in an include file mpi.h.

Almost all C functions return an error code. The successful return code will be MPI_SUCCESS, but failure return codes are implementation dependent.

Type declarations are provided for handles to each category of opaque objects.

Array arguments are indexed from zero.

Logical flags are integers with value 0 meaning "false" and a non-zero value meaning "true."

Choice arguments are pointers of type void*.

2.6.4 Functions and Macros

An implementation is allowed to implement MPI_WTIME, PMPI_WTIME, MPI_WTICK, PMPI_WTICK, MPI_AINT_ADD, PMPI_AINT_ADD, MPI_AINT_DIFF, PMPI_AINT_DIFF, and the handle-conversion functions (MPI_Group_f2c, etc.) in Section 19.2.4, and no others, as macros in C.

Advice to implementors. Implementors should document which routines are implemented as macros. (End of advice to implementors.)

Advice to users. If these routines are implemented as macros, they will not work with the MPI profiling interface. (End of advice to users.)

2.7 Processes

An MPI program consists of autonomous processes, executing their own code, in an MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible.

This document specifies the behavior of a parallel program assuming that only MPI calls are used. The interaction of an MPI program with other possible means of communication, I/O, and process management is not specified. Unless otherwise stated in the specification of the standard, MPI places no requirements on the result of its interaction

Unofficial Draft for Comment Only

with external mechanisms that provide similar or equivalent functionality. This includes,
 but is not limited to, interactions with external mechanisms for process control, shared and
 remote memory access, file system access and control, interprocess communication, process
 signaling, and terminal I/O. High quality implementations should strive to make the results
 of such interactions intuitive to users, and attempt to document restrictions where deemed
 necessary.

Advice to implementors. Implementations that support such additional mechanisms for functionality supported within MPI are expected to document how these interact with MPI. (End of advice to implementors.)

- The interaction of MPI and threads is defined in Section 11.6.
- 12 13 14

15

7

8

9

10 11

2.8 Error Handling

16MPI provides the user with reliable message transmission. A message sent is always re-17ceived correctly, and the user does not need to check for transmission errors, time-outs, 18 or other error conditions. In other words, MPI does not provide mechanisms for dealing 19with transmission failures in the communication system. If the MPI implementation is 20built on an unreliable underlying mechanism, then it is the job of the implementor of the 21MPI subsystem to insulate the user from this unreliability, and to reflect only unrecoverable 22transmission failures. Whenever possible, such failures will be reflected as errors in the 23relevant communication call.

²⁴ Similarly, MPI itself provides no mechanisms for handling MPI process failures, that
 ²⁵ is, when an MPI process unexpectedly and permanently stops communicating (e.g., a software or hardware crash results in an MPI process terminating unexpectedly).

27Of course, MPI programs may still be erroneous. A **program error** can occur when 28 an MPI call is made with an incorrect argument (non-existing destination in a send oper-29ation, buffer too small in a receive operation, etc.). This type of error would occur in any 30 implementation. In addition, a **resource error** may occur when a program exceeds the 31 amount of available system resources (number of pending messages, system buffers, etc.). 32 The occurrence of this type of error depends on the amount of available resources in the 33 system and the resource allocation mechanism used: this may differ from system to system. 34A high-quality implementation will provide generous limits on the important resources so 35 as to alleviate the portability problem this represents.

36 In C and Fortran, almost all MPI calls return a code that indicates successful completion 37 of the operation. Whenever possible, MPI calls return an error code if an error occurred 38 during the call. By default, an error detected during the execution of the MPI library 39 causes the parallel computation to abort, except for file operations. However, MPI provides 40mechanisms for users to change this default and to handle recoverable errors. The user may 41 specify that no error is fatal, and handle error codes returned by MPI calls by himself or 42herself. Also, the user may provide his or her own error-handling routines, which will be 43invoked whenever an MPI call returns abnormally. The MPI error handling facilities are 44described in Section 9.3.

⁴⁵ Several factors limit the ability of MPI calls to return with meaningful error codes
 ⁴⁶ when an error occurs. MPI may not be able to detect some errors; other errors may be too
 ⁴⁷ expensive to detect in normal execution mode; finally some errors may be "catastrophic"
 ⁴⁸ and may prevent MPI from returning control to the caller. On the other hand, some errors

Unofficial Draft for Comment Only

may be detected after the associated operation has completed; some errors may not have a communicator, window, or file on which an error may be raised. In such cases, these errors will be raised on the communicator MPI_COMM_SELF. When MPI_COMM_SELF is not initialized (i.e., before MPI_INIT / MPI_INIT_THREAD or after MPI_FINALIZE) the error raises the **initial error handler** (set during the launch operation, see 11.8.4).

An example of such a case arises because of the nature of asynchronous communications: MPI calls may initiate operations that continue asynchronously after the call returned. Thus, the operation may return with a code indicating successful completion, yet later cause an error to be raised. If there is a subsequent call that relates to the same operation (e.g., a call that verifies that an asynchronous operation has completed) then the error argument associated with this call will be used to indicate the nature of the error. In a few cases, the error may occur after all calls that relate to the operation have completed, so that no error value can be used to indicate the nature of the error (e.g., an error on the receiver in a send with the ready mode).

This document does not specify the state of a computation after an erroneous MPI call has occurred. The desired behavior is that a relevant error code be returned, and the effect of the error be localized to the greatest possible extent. E.g., it is highly desirable that an erroneous receive call will not cause any part of the receiver's memory to be overwritten, beyond the area specified for receiving the message.

Implementations may go beyond this document in supporting in a meaningful manner MPI calls that are defined here to be erroneous. For example, MPI specifies strict type matching rules between matching send and receive operations: it is erroneous to send a floating point variable and receive an integer. Implementations may go beyond these type matching rules, and provide automatic type conversion in such situations. It will be helpful to generate warnings for such non-conforming behavior.

MPI defines a way for users to create new error codes as defined in Section 9.5.

2.9 Implementation Issues

There are a number of areas where an MPI implementation may interact with the operating environment and system. While MPI does not mandate that any services (such as signal handling) be provided, it does strongly suggest the behavior to be provided if those services are available. This is an important point in achieving portability across platforms that provide the same set of services.

2.9.1 Independence of Basic Runtime Routines

MPI programs require that library routines that are part of the basic language environment (such as write in Fortran and printf and malloc in ISO C) and are executed after MPI_INIT and before MPI_FINALIZE operate independently and that their *completion* is independent of the action of other processes in an MPI program.

Note that this in no way prevents the creation of library routines that provide parallel services whose operation is collective. However, the following program is expected to complete in an ISO C environment regardless of the size of MPI_COMM_WORLD (assuming that printf is available at the executing nodes).

```
int rank;
MPI_Init((void *)0, (void *)0);
```

1

 $\mathbf{2}$

3

4

5

6 7

8 9

10

11

12

13 14

15

16

17

18

19

20

21

22

23 24

25

26

27 28

29 30

32 33 34

35 36

37

38 39

40 41

42

43 44

 $45 \\ 46$

47

48

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
if (rank == 0) printf("Starting program\n");
```

```
MPI_Finalize();
```

The corresponding Fortran programs are also expected to complete.

An example of what is *not* required is any particular ordering of the action of these routines when called by several tasks. For example, MPI makes neither requirements nor recommendations for the output from the following program (again assuming that I/O is available at the executing nodes).

¹⁰ MPI_Comm_rank(MPI_COMM_WORLD, &rank); ¹¹ printf("Output from task rank %d\n", rank);

¹³ In addition, calls that fail because of resource exhaustion or other error are not con-¹⁴ sidered a violation of the requirements here (however, they are required to complete, just ¹⁵ not to complete successfully).

16

¹⁷ 2.9.2 Interaction with Signals

¹⁹ MPI does not specify the interaction of processes with signals and does not require that MPI ²⁰ be signal safe. The implementation may reserve some signals for its own use. It is required ²¹ that the implementation document which signals it uses, and it is strongly recommended ²² that it not use SIGALRM, SIGFPE, or SIGIO. Implementations may also prohibit the use of ²³ MPI calls from within signal handlers.

In multithreaded environments, users can avoid conflicts between signals and the MPI library by catching signals only on threads that do not execute MPI calls. High quality single-threaded implementations will be signal safe: an MPI call suspended by a signal will resume and complete normally after the signal is handled.

28 29

30

2.10 Examples

The examples in this document are for illustration purposes only. They are not intended to specify the standard. Furthermore, the examples have not been carefully checked or verified.

3

4

Chapter 3

Point-to-Point Communication

3.1Introduction

Sending and receiving of messages by processes is the basic MPI communication mechanism. The basic point-to-point communication operations are send and receive. Their use is illustrated in the example below.

```
20
#include "mpi.h"
                                                                                   21
int main(int argc, char *argv[])
                                                                                   22
{
                                                                                   23
  char message[20];
  int myrank;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  if (myrank == 0)
                       /* code for process zero */
                                                                                   29
  ſ
      strcpy(message,"Hello, there")
      MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else if (myrank == 1) /* code for process one */
                                                                                   34
  Ł
                                                                                   35
      MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
                                                                                   36
      printf("received :%s:\n", message);
                                                                                   37
  }
 MPI_Finalize()
  return 0;
}
```

42In this example, process zero (myrank = 0) sends a message to process one using the send operation MPI_SEND. The operation specifies a send buffer in the sender memory 43 44from which the message data is taken. In the example above, the send buffer consists of the storage containing the variable **message** in the memory of process zero. The location, size and type of the send buffer are specified by the first three parameters of the send operation. The message sent will contain the 13 characters of this variable. In addition, the send operation associates an **envelope** with the message. This envelope specifies the

1 2 3

6

1516

17

18

19

24

25

26

27

28

30

31

32

33

38

39

40

41

4546

47

1 message destination and contains distinguishing information that can be used by the **receive** $\mathbf{2}$ operation to select a particular message. The last three parameters of the send operation, 3 along with the rank of the sender, specify the envelope for the message sent. Process one 4 (myrank = 1) receives this message with the receive operation MPI_RECV. The message to 5be received is selected according to the value of its envelope, and the message data is stored 6 into the **receive buffer**. In the example above, the receive buffer consists of the storage 7containing the string message in the memory of process one. The first three parameters 8 of the receive operation specify the location, size and type of the receive buffer. The next 9 three parameters are used for selecting the incoming message. The last parameter is used 10 to return information on the message just received.

¹¹ The next sections describe the blocking send and receive operations. We discuss send, ¹² receive, blocking communication semantics, type matching requirements, type conversion in ¹³ heterogeneous environments, and more general communication modes. Nonblocking com-¹⁴ munication is addressed next, followed by probing and canceling a message, channel-like ¹⁵ constructs and send-receive operations, ending with a description of the "dummy" process, ¹⁶ MPI_PROC_NULL.

3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
```

26	IN	buf	initial address of send buffer (choice)
27 28	IN	count	number of elements in send buffer (non-negative integer)
29	INI	datatives	
30	IN	datatype	datatype of each send buffer element (handle)
31	IN	dest	rank of destination (integer)
32 33	IN	tag	message tag (integer)
34	IN	comm	communicator (handle)

```
36 C binding
```

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
```

```
<sup>39</sup> Fortran 2008 binding
```

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER, INTENT(IN) :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
```

```
<sup>48</sup> MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
```

17 18

19 20

21

22 23 24

25

<type> BUF(*)</type>					
INTEGER COUNT,	DATATYPE,	DEST,	TAG,	COMM,	IERROR

The blocking semantics of this call are described in Section 3.4.

3.2.2 Message Data

The send buffer specified by the MPI_SEND operation consists of count successive entries of the type indicated by datatype, starting with the entry at address buf. Note that we specify the message length in terms of number of *elements*, not number of *bytes*. The former is machine independent and closer to the application level.

The data part of the message consists of a sequence of **count** values, each of the type indicated by **datatype**. **count** may be zero, in which case the data part of the message is empty. The basic datatypes that can be specified for message data values correspond to the basic datatypes of the host language. Possible values of this argument for Fortran and the corresponding Fortran types are listed in Table 3.1.

Fortran datatype
INTEGER
REAL
DOUBLE PRECISION
COMPLEX
LOGICAL
CHARACTER(1)

Table 3.1: Predefined MPI datatypes corresponding to Fortran datatypes

Possible values for this argument for C and the corresponding C types are listed in Table 3.2.

The datatypes MPI_BYTE and MPI_PACKED do not correspond to a Fortran or C datatype. A value of type MPI_BYTE consists of a byte (8 binary digits). A byte is uninterpreted and is different from a character. Different machines may have different representations for characters, or may use more than one byte to represent characters. On the other hand, a byte has the same binary value on all machines. The use of the type MPI_PACKED is explained in Section 5.2.

MPI requires support of these datatypes, which match the basic datatypes of Fortran and ISO C. Additional MPI datatypes should be provided if the host language has additional data types¹: MPI_DOUBLE_COMPLEX for double precision complex in Fortran declared to be of type DOUBLE COMPLEX; MPI_REAL2, MPI_REAL4, MPI_REAL8, and MPI_REAL16 for Fortran reals, declared to be of type REAL*2, REAL*4, REAL*8, and REAL*16, respectively; MPI_INTEGER1, MPI_INTEGER2, MPI_INTEGER4, and MPI_INTEGER8 for Fortran integers, declared to be of type INTEGER*1, INTEGER*2, INTEGER*4, and INTEGER*8, respectively; MPI_COMPLEX4, MPI_COMPLEX8,

Unofficial Draft for Comment Only

¹These types, such as DOUBLE COMPLEX and INTEGER*4, are not specified by any Fortran standard but are extensions commonly accepted by Fortran compilers.

1	MPI datatype	C datatype
2	MPI_CHAR	char
3		(treated as printable character)
4	MPI_SHORT	signed short int
5	MPI_INT	signed int
6	MPI_LONG	signed long int
7	MPI_LONG_LONG_INT	signed long long int
8	MPI_LONG_LONG (as a synonym)	signed long long int
9	MPI_SIGNED_CHAR	signed char
10		(treated as integral value)
11	MPI_UNSIGNED_CHAR	unsigned char
12		(treated as integral value)
13	MPI_UNSIGNED_SHORT	unsigned short int
14	MPI_UNSIGNED	unsigned int
15	MPI_UNSIGNED_LONG	unsigned long int
16	MPI_UNSIGNED_LONG_LONG	unsigned long long int
17	MPI_FLOAT	float
18	MPI_DOUBLE	double
19	MPI_LONG_DOUBLE	long double
20	MPI_WCHAR	wchar_t
21		(defined in <stddef.h>)</stddef.h>
22		(treated as printable character)
23	MPI_C_BOOL	_Bool
24	MPI_INT8_T	int8_t
25	MPI_INT16_T	int16_t
26	MPI_INT32_T	int32_t
27	MPI_INT64_T	int64_t
28	MPI_UINT8_T	uint8_t
29	MPI_UINT16_T	uint16_t
30	MPI_UINT32_T	uint32_t
31	MPI_UINT64_T	uint64_t
32	MPI_C_COMPLEX	float _Complex
33	MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
34	MPI_C_DOUBLE_COMPLEX	double _Complex
35	MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
36	MPI_BYTE	
37	MPI_PACKED	
38]
39		
40	Table 3.2: Predefined MPI datatypes of	corresponding to C datatypes

MPI_COMPLEX16, and MPI_COMPLEX32 for complex numbers in Fortran declared to be of type COMPLEX*4, COMPLEX*8, COMPLEX*16, and COMPLEX*32, respectively; etc.

Rationale. One goal of the design is to allow for MPI to be implemented as a library, with no need for additional preprocessing or compilation. Thus, one cannot assume that a communication call has information on the datatype of variables in the communication buffer; this information must be supplied by an explicit argument.

MPI datatype	C datatype	Fortran datatype
MPI_AINT	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)
MPI_OFFSET	MPI_Offset	INTEGER (KIND=MPI_OFFSET_KIND)
MPI_COUNT	MPI_Count	INTEGER (KIND=MPI_COUNT_KIND)

Table 3.3: Predefined MPI datatypes corresponding to both C and Fortran datatypes

The need for such datatype information will become clear in Section 3.3.2. (End of rationale.)

The datatypes MPI_AINT, MPI_OFFSET, and MPI_COUNT correspond to the MPI-defined C types MPI_Aint, MPI_Offset, and MPI_Count and their Fortran equivalents INTEGER (KIND=MPI_ADDRESS_KIND), INTEGER (KIND=MPI_OFFSET_KIND), and INTEGER (KIND=MPI_COUNT_KIND). This is described in Table 3.3. All predefined datatype handles are available in all language bindings. See Sections 19.2.6 and 19.2.10 on page 785 and 792 for information on interlanguage communication with these types.

If there is an accompanying C++ compiler then the datatypes in Table 3.4 are also supported in C and Fortran.

C++ datatype
bool
<pre>std::complex<float></float></pre>
std::complex <double></double>
<pre>std::complex<long double=""></long></pre>

Table 3.4: Predefined MPI datatypes corresponding to C++ datatypes

3.2.3 Message Envelope

In addition to the data part, messages carry information that can be used to distinguish messages and selectively receive them. This information consists of a fixed number of fields, which we collectively call the **message envelope**. These fields are

source destination tag

communicator

The message source is implicitly determined by the identity of the message sender. The other fields are specified by arguments in the send operation.

The message destination is specified by the dest argument.

The integer-valued message tag is specified by the tag argument. This integer can be 44used by the program to distinguish different types of messages. The range of valid tag values is $0, \ldots, \mathsf{UB}$, where the value of UB is implementation dependent. It can be found by querying the value of the attribute MPI_TAG_UB, as described in Chapter 9. MPI requires that UB be no less than 32767.

Unofficial Draft for Comment Only

2

8

9

10 11

12

13

14

15

16

17

18

28 29 30

31

32

33

34 35

36

37

38

39 40

41

42

43

45

46

47

The comm argument specifies the communicator that is used for the send operation. Communicators are explained in Chapter 7; below is a brief summary of their usage.

³ A communicator specifies the communication context for a communication operation. ⁴ Each communication context provides a separate "communication universe": messages are ⁵ always received within the context they were sent, and messages sent in different contexts ⁶ do not interfere.

⁷ The communicator also specifies the set of processes that share this communication ⁸ context. This **process group** is ordered and processes are identified by their rank within ⁹ this group. Thus, the range of valid values for **dest** is $0, \ldots, n-1 \cup \{\text{MPI}_{PROC_NULL}\}$, where ¹⁰ n is the number of processes in the group. (If the communicator is an inter-communicator, ¹¹ then destinations are identified by their rank in the remote group. See Chapter 7.)

¹² When using the World Model (see Section 11.1), a predefined communicator
 ¹³ MPI_COMM_WORLD is provided by MPI. It allows communication with all processes that
 ¹⁴ are accessible after MPI initialization and processes are identified by their rank in the group
 ¹⁵ of MPI_COMM_WORLD.

Advice to users. Users that are comfortable with the notion of a flat name space
 for processes, and a single communication context, as offered by most existing com munication libraries, need only use the World Model for MPI initialization, and the
 predefined variable MPI_COMM_WORLD as the comm argument. This will allow com munication with all the processes available at initialization time.

²² Users may define new communicators, as explained in Chapter 7. Communicators ²³ provide an important encapsulation mechanism for libraries and modules. They allow ²⁴ modules to have their own disjoint communication universe and their own process ²⁵ numbering scheme. (*End of advice to users.*)

Advice to implementors. The message envelope would normally be encoded by a fixed-length message header. However, the actual encoding is implementation dependent. Some of the information (e.g., source or destination) may be implicit, and need not be explicitly carried by messages. Also, processes may be identified by relative ranks, or absolute ids, etc. (End of advice to implementors.)

- 3.2.4 Blocking Receive
- The syntax of the blocking receive operation is given below.

1

 $\mathbf{2}$

16

27

28

29

30

31

32 33

34

MPI_REC	CV(buf, count, dataty	vpe, source, tag, comm, status)	1
OUT	buf	initial address of receive buffer (choice)	2
IN	count	number of elements in receive buffer (non-negative integer)	3 4 5
IN	datatype	datatype of each receive buffer element (handle)	6
IN	source	rank of source or MPI_ANY_SOURCE (integer)	7
IN	tag	message tag or MPI_ANY_TAG (integer)	8 9
IN	comm	communicator (handle)	9 10
			11
OUT	status	status object (Status)	12
a 1 · 1 ·			13
C bindi	0	int count MDT Detetune detetune int counce	14
INC MPI_		<pre>int count, MPI_Datatype datatype, int source, PI_Comm comm, MPI_Status *status)</pre>	15
	0		16
	2008 binding		17
		atype, source, tag, comm, status, ierror)	18 19
	E(*), DIMENSION(.		20
		<pre>:: count, source, tag INTENT(IN) :: datatype</pre>	21
	E(MPI_Comm), INTE		22
	E(MPI_Status) ::		23
		NTENT(OUT) :: ierror	24
			25
Fortran	0	ATTADE COUDCE TAC COMM CTATUC LEDDOD)	26
	De> BUF(*)	ATYPE, SOURCE, TAG, COMM, STATUS, IERROR)	27
		YPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),	28
	IERROR		29
-			30 31
		of this call are described in Section 3.4.	32
		at address buf . The length of the received message must	33
		length of the receive buffer. An overflow error occurs if all	34

incoming data does not fit, without truncation, into the receive buffer. If a message that is shorter than the receive buffer arrives, then only those locations

If a message that is shorter than the receive buffer arrives, then only those locations corresponding to the (shorter) message are modified.

Advice to users. The MPI_PROBE function described in Section 3.8 can be used to receive messages of unknown length. (*End of advice to users.*)

Advice to implementors. Even though no specific behavior is mandated by MPI for erroneous programs, the recommended handling of overflow situations is to return in status information about the source and tag of the incoming message. The receive operation will return an error code. A quality implementation will also ensure that no memory that is outside the receive buffer will ever be overwritten.

In the case of a message shorter than the receive buffer, MPI is quite strict in that it allows no modification of the other locations. A more lenient statement would allow 48

Unofficial Draft for Comment Only

35 36

37

38

39

40 41

42

43

44

45

for some optimizations but this is not allowed. The implementation must be ready to end a copy into the receiver memory exactly at the end of the receive buffer, even if it is an odd address. (*End of advice to implementors.*)

The selection of a message by a receive operation is governed by the value of the 56 message envelope. A message can be received by a receive operation if its envelope matches the source, tag and comm values specified by the receive operation. The receiver may specify $\overline{7}$ a wildcard MPI_ANY_SOURCE value for source, and/or a wildcard MPI_ANY_TAG value for 8 tag, indicating that any source and/or tag are acceptable. It cannot specify a wildcard value 9 10 for comm. Thus, a message can be received by a receive operation only if it is addressed to the receiving process, has a matching communicator, has matching source unless source 11= MPI_ANY_SOURCE in the pattern, and has a matching tag unless tag = MPI_ANY_TAG in 12the pattern. 13

The message tag is specified by the tag argument of the receive operation. The argu-14ment source, if different from MPI_ANY_SOURCE, is specified as a rank within the process 1516group associated with that same communicator (remote process group, for intercommunicators). Thus, the range of valid values for the source argument is $\{0, \ldots, n-1\} \cup$ 17 $\{MPI_ANY_SOURCE\} \cup \{MPI_PROC_NULL\}, where n is the number of processes in this group.$ 18 Note the asymmetry between send and receive operations: A receive operation may 19accept messages from an arbitrary sender, on the other hand, a send operation must specify 2021a unique receiver. This matches a "push" communication mechanism, where data transfer is effected by the sender (rather than a "pull" mechanism, where data transfer is effected 22by the receiver). 23

Source = destination is allowed, that is, a process can send a message to itself. However,
 it is unsafe to do so with the blocking send and receive operations described above, since
 this may lead to deadlock. See Section 3.5.

27 28

29

30

 31

32 33

34

Advice to implementors. Message context and other communicator information can be implemented as an additional tag field. It differs from the regular message tag in that wild card matching is not allowed on this field, and that value setting for this field is controlled by communicator manipulation functions. (*End of advice to implementors.*)

The use of $dest = MPI_PROC_NULL$ or $source = MPI_PROC_NULL$ to define a "dummy" destination or source in any send or receive call is described in Section 3.10.

35 36 37

3.2.5 Return Status

The source or tag of a received message may not be known if wildcard values were used in the receive operation. Also, if multiple requests are completed by a single MPI function (see Section 3.7.5), a distinct error code may need to be returned for each request. The information is returned by the status argument of MPI_RECV. The type of status is MPIdefined. Status variables need to be explicitly allocated by the user, that is, they are not system objects.

⁴⁴ In C, status is a structure that contains three fields named MPI_SOURCE, MPI_TAG, ⁴⁵ and MPI_ERROR; the structure may contain additional fields. Thus,

status.MPI_SOURCE, status.MPI_TAG and status.MPI_ERROR contain the source, tag, and
 error code, respectively, of the received message.

Unofficial Draft for Comment Only

1

2

3

In Fortran with USE mpi or INCLUDE 'mpif.h', status is an array of INTEGERS of size MPI_STATUS_SIZE. The constants MPI_SOURCE, MPI_TAG and MPI_ERROR are the indices of the entries that store the source, tag and error fields. Thus, status(MPI_SOURCE), status(MPI_TAG) and status(MPI_ERROR) contain, respectively, the source, tag and error code of the received message.

With Fortran USE mpi_f08, status is defined as the Fortran BIND(C) derived type TYPE(MPI_Status) containing three public INTEGER fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR. TYPE(MPI_Status) may contain additional, implementation-specific fields. Thus, status%MPI_SOURCE, status%MPI_TAG and status%MPI_ERROR contain the source, tag, and error code of a received message respectively. Additionally, within both the mpi and the mpi_f08 modules, the constants MPI_STATUS_SIZE, MPI_SOURCE, MPI_TAG, MPI_ERROR, and TYPE(MPI_Status) are defined to allow conversion between both status representations. Conversion routines are provided in Section 19.2.5.

Rationale. The Fortran TYPE(MPI_Status) is defined as a BIND(C) derived type so that it can be used at any location where the status integer array representation can be used, e.g., in user defined common blocks. (*End of rationale.*)

Rationale. It is allowed to have the same name (e.g., MPI_SOURCE) defined as a constant (e.g., Fortran parameter) and as a field of a derived type. (*End of rationale.*)

In general, message-passing calls do not modify the value of the error code field of status variables. This field may be updated only by the functions in Section 3.7.5 which return multiple statuses. The field is updated if and only if such function returns with an error code of MPI_ERR_IN_STATUS.

Rationale. The error field in status is not needed for calls that return only one status, such as MPI_WAIT, since that would only duplicate the information returned by the function itself. The current design avoids the additional overhead of setting it, in such cases. The field is needed for calls that return multiple statuses, since each request may have had a different failure. (*End of rationale.*)

The status argument also returns information on the length of the message received. However, this information is not directly available as a field of the status variable and a call to MPI_GET_COUNT is required to "decode" this information.

MPI_GET_COUNT(status, datatype, count)

IN	status	return status of receive operation (Status)
IN	datatype	datatype of each receive buffer entry (handle)
OUT	count	number of received entries (integer)

C binding

Fortran 2008 binding

MPI_Get_count(status, datatype, count, ierror)

Unofficial Draft for Comment Only

 $\overline{7}$

1 2 3 4	TYPE(MPI_Status), INTENT(IN) :: status TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(OUT) :: count INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5 6 7 8	Fortran binding MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
9 10 11 12 13 14	Returns the number of entries received. (Again, we count <i>entries</i> , each of type <i>datatype</i> , not <i>bytes</i> .) The datatype argument should match the argument provided by the receive call that set the status variable. If the number of entries received exceeds the limits of the count parameter, then MPI_GET_COUNT sets the value of count to MPI_UNDEFINED. There are other situations where the value of count can be set to MPI_UNDEFINED; see Section 5.1.11.
15 16 17 18 19 20	<i>Rationale.</i> Some message-passing libraries use INOUT count, tag and source arguments, thus using them both to specify the selection criteria for incoming messages and return the actual envelope values of the received message. The use of a separate status argument prevents errors that are often attached with INOUT argument (e.g., using the MPI_ANY_TAG constant as the tag in a receive). Some libraries use calls that refer implicitly to the "last message received." This is not thread safe.
21 22 23 24 25 26 27	The datatype argument is passed to MPI_GET_COUNT so as to improve performance. A message might be received without counting the number of elements it contains, and the count value is often not needed. Also, this allows the same function to be used after a call to MPI_PROBE or MPI_IPROBE. With a status from MPI_PROBE or MPI_IPROBE, the same datatypes are allowed as in a call to MPI_RECV to receive this message. (<i>End of rationale.</i>)
28 29 30 31	The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI_UNDEFINED is returned.
32 33 34 35 36 37	<i>Rationale.</i> Zero-length datatypes may be created in a number of cases. An important case is MPI_TYPE_CREATE_DARRAY, where the definition of the particular darray results in an empty block on some MPI process. Programs written in an SPMD style will not check for this special case and may want to use MPI_GET_COUNT to check the status. (<i>End of rationale.</i>)
38 39 40 41 42	Advice to users. The buffer size required for the receive can be affected by data conversions and by the stride of the receive datatype. In most cases, the safest approach is to use the same datatype with MPI_GET_COUNT and the receive. (<i>End of advice to users.</i>)
43 44 45 46 47 48	All send and receive operations use the buf, count, datatype, source, dest, tag, comm, and status arguments in the same way as the blocking MPI_SEND and MPI_RECV operations described in this section.

3.2.6 Passing MPI_STATUS_IGNORE for Status

Every call to MPI_RECV includes a status argument, wherein the system can return details about the message received. There are also a number of other MPI calls where status is returned. An object of type MPI_Status is not an MPI opaque object; its structure is declared in mpi.h and mpif.h, and it exists in the user's program. In many cases, application programs are constructed so that it is unnecessary for them to examine the status fields. In these cases, it is a waste for the user to allocate a status object, and it is particularly wasteful for the MPI implementation to fill in fields in this object.

To cope with this problem, there are two predefined constants, MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE, which when passed to a receive, probe, wait, or test function, inform the implementation that the status fields are not to be filled in. Note that MPI_STATUS_IGNORE is not a special type of MPI_Status object; rather, it is a special value for the argument. In C one would expect it to be NULL, not the address of a special MPI_Status.

MPI_STATUS_IGNORE, and the array version MPI_STATUSES_IGNORE, can be used everywhere a status argument is passed to a receive, wait, or test function. MPI_STATUS_IGNORE cannot be used when status is an IN argument. Note that in Fortran MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE are objects like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

In general, this optimization can apply to all functions for which status or an array of statuses is an OUT argument. Note that this converts status into an INOUT argument. The functions that can be passed MPI_STATUS_IGNORE are all the various forms of MPI_RECV, MPI_PROBE, MPI_TEST, and MPI_WAIT, as well as MPI_REQUEST_GET_STATUS. When an array is passed, as in the MPI_{TEST|WAIT}{ALL|SOME} functions, a separate constant, MPI_STATUSES_IGNORE, is passed for the array argument. It is possible for an MPI function to return MPI_ERR_IN_STATUS even when MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE has been passed to that function.

MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE are not required to have the same values in C and Fortran.

It is not allowed to have some of the statuses in an array of statuses for MPI_{TEST|WAIT}{ALL|SOME} functions set to MPI_STATUS_IGNORE; one either specifies ignoring *all* of the statuses in such a call with MPI_STATUSES_IGNORE, or *none* of them by passing normal statuses in all positions in the array of statuses.

3.2.7 Send-Receive

The **send-receive** operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly (for example, even processes send, then receive, odd processes receive first, then send) so as to prevent cyclic dependencies that may lead to deadlock. When a send-receive operation is used, the communication subsystem takes care of these issues. The send-receive operation can be used in conjunction with the functions described in Chapter 8 in order to perform shifts on various logical topologies. Also, a send-receive operation is useful for implementing remote procedure calls.

Unofficial Draft for Comment Only

 24

A message sent by a send-receive operation can be received by a regular receive operation or probed by a probe operation; a send-receive operation can receive a message sent by a regular send operation.

5 6	MPI_SEN	DRECV(sendbuf, sendcount, se source, recvtag, comm, s	ndtype, dest, sendtag, recvbuf, recvcount, recvtype, tatus)
7 8	IN	sendbuf	initial address of send buffer (choice)
9 10 11	IN	sendcount	number of elements in send buffer (non-negative integer)
12	IN	sendtype	type of elements in send buffer (handle)
13	IN	dest	rank of destination (integer)
14 15	IN	sendtag	send tag (integer)
16	OUT	recvbuf	initial address of receive buffer (choice)
17 18	IN	recvcount	number of elements in receive buffer (non-negative integer)
19 20	IN	recvtype	type of elements receive buffer element (handle)
21	IN	source	rank of source or MPI_ANY_SOURCE (integer)
22	IN	recvtag	receive tag or MPI_ANY_TAG (integer)
23 24	IN	comm	communicator (handle)
25	OUT	status	status object (Status)
26 27 28 29 30 31	C binding int MPI_S	Sendrecv(const void *send int dest, int sendta	buf, int sendcount, MPI_Datatype sendtype, ng, void *recvbuf, int recvcount, ne, int source, int recvtag, MPI_Comm comm,
32 33 34 35 36 37 38 39 40 41 42 43	MPI_Sendr TYPE(INTEC TYPE(TYPE(TYPE)	recvcount, recvtype, (*), DIMENSION(), INTEN	unt, dest, sendtag, recvcount, source,) :: sendtype, recvtype vbuf comm
44	Fortran b	binding	

```
38
```

1

 $\mathbf{2}$

3

4 5

INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

Execute a blocking send and receive operation. Both send and receive use the same communicator, but possibly different tags. The send buffer and receive buffers must be disjoint, and may have different lengths and datatypes.

The semantics of a send-receive operation is what would be obtained if the caller forked two concurrent threads, one to execute the send, and one to execute the receive, followed by a join of these two threads.

MPI_SEND	RECV_REPLACE(buf, co status)	unt, datatype, dest, sendtag, source, recvtag, comm,		
INOUT	buf	initial address of send and receive buffer (choice)		
IN	count	number of elements in send and receive buffer (non-negative integer)		
IN	datatype	type of elements in send and receive buffer (handle)		
IN	dest	rank of destination (integer)		
IN	sendtag	send message tag (integer)		
IN	source	rank of source or MPI_ANY_SOURCE (integer)		
IN	recvtag	receive message tag or MPI_ANY_TAG (integer)		
IN	comm	communicator (handle)		
OUT	status	status object (Status)		
C binding				

Fortran 2008 binding MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag, comm, status, ierror) TYPE(*), DIMENSION(..) :: buf INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

 $\mathbf{2}$

Execute a blocking send and receive. The same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received.

Advice to implementors. Additional intermediate buffering is needed for the "replace" variant. (*End of advice to implementors.*)

3.3 Data Type Matching and Data Conversion

3.3.1 Type Matching Rules

¹¹ One can think of message transfer as consisting of the following three phases.

- 1. Data is pulled out of the send buffer and a message is assembled.
- 2. A message is transferred from sender to receiver.
- 14 15 16

3. Data is pulled from the incoming message and disassembled into the receive buffer.

Type matching has to be observed at each of these three phases: The type of each variable in the sender buffer has to match the type specified for that entry by the send operation; the type specified by the send operation has to match the type specified by the receive operation; and the type of each variable in the receive buffer has to match the type specified for that entry by the receive operation. A program that fails to observe these three rules is erroneous.

To define type matching more precisely, we need to deal with two issues: matching of types of the host language with types specified in communication operations; and matching of types at sender and receiver.

The types of a send and receive match (phase two) if both operations use identical names. That is, MPI_INTEGER matches MPI_INTEGER, MPI_REAL matches MPI_REAL, and so on. There is one exception to this rule, discussed in Section 5.2: the type MPI_PACKED can match any other type.

The type of a variable in a host program matches the type specified in the commu-30 nication operation if the datatype name used by that operation corresponds to the basic 31 type of the host program variable. For example, an entry with type name MPI_INTEGER 32 matches a Fortran variable of type INTEGER. A table giving this correspondence for Fortran 33 34and C appears in Section 3.2.2. There are two exceptions to this last rule: an entry with type name MPI_BYTE or MPI_PACKED can be used to match any byte of storage (on a byte-35 addressable machine), irrespective of the datatype of the variable that contains this byte. 36 The type MPI_PACKED is used to send data that has been explicitly packed, or receive data 37 that will be explicitly unpacked, see Section 5.2. The type MPI_BYTE allows one to transfer 38the binary value of a byte in memory unchanged. 39

40 41

42

43

44

To summarize, the type matching rules fall into the three categories below.

- Communication of typed values (e.g., with datatype different from MPI_BYTE), where the datatypes of the corresponding entries in the sender program, in the send call, in the receive call and in the receiver program must all match.
- Communication of untyped values (e.g., of datatype MPI_BYTE), where both sender and receiver use the datatype MPI_BYTE. In this case, there are no requirements on the types of the corresponding entries in the sender and the receiver programs, nor is it required that they be the same.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5 6 7

> 8 9

10

12

```
• Communication involving packed data, where MPI_PACKED is used.
    The following examples illustrate the first two cases.
Example 3.1 Sender and receiver specify matching types.
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
   CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
   CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
END IF
    This code is correct if both a and b are real arrays of size \geq 10. (In Fortran, it might be
correct to use this code even if a or b have size < 10: e.g., when a(1) can be equivalenced
to an array with ten reals.)
Example 3.2 Sender and receiver do not specify matching types.
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
   CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
   CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
    This code is erroneous, since sender and receiver do not provide matching datatype
arguments.
Example 3.3 Sender and receiver specify communication of untyped values.
```

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

This code is correct, irrespective of the type and size of **a** and **b** (unless this results in an out of bounds memory access).

Advice to users. If a buffer of type MPI_BYTE is passed as an argument to MPI_SEND, then MPI will send the data stored at contiguous locations, starting from the address indicated by the buf argument. This may have unexpected results when the data layout is not as a casual user would expect it to be. For example, some Fortran compilers implement variables of type CHARACTER as a structure that contains the character length and a pointer to the actual string. In such an environment, sending and receiving a Fortran CHARACTER variable using the MPI_BYTE type will not have the anticipated result of transferring the character string. For this reason, the user is advised to use typed communications whenever possible. (*End of advice to users.*)

Unofficial Draft for Comment Only

1

2

10

11

12

13

14

15 16

17 18

19

20

21

22

23

24 25

26

27 28

29

30

31

32

33

34

35 36

37

38 39

40

41

42

43

44

45

46

47

```
1
      Type MPI_CHARACTER
2
      The type MPI_CHARACTER matches one character of a Fortran variable of type
3
      CHARACTER, rather than the entire character string stored in the variable. Fortran variables
4
      of type CHARACTER or substrings are transferred as if they were arrays of characters. This
5
      is illustrated in the example below.
6
7
      Example 3.4 Transfer of Fortran CHARACTERs.
8
9
      CHARACTER*10 a
10
      CHARACTER*10 b
11
12
     CALL MPI_COMM_RANK(comm, rank, ierr)
13
      IF (rank .EQ. 0) THEN
14
         CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
15
     ELSE IF (rank .EQ. 1) THEN
16
         CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status,
                                                                                   ierr)
17
     END IF
18
19
          The last five characters of string b at process 1 are replaced by the first five characters
20
      of string a at process 0.
21
22
           Rationale. The alternative choice would be for MPI_CHARACTER to match a character
23
           of arbitrary length. This runs into problems.
24
           A Fortran character variable is a constant length string, with no special termina-
25
           tion symbol. There is no fixed convention on how to represent characters, and how
26
           to store their length. Some compilers pass a character argument to a routine as a
27
           pair of arguments, one holding the address of the string and the other holding the
28
           length of string. Consider the case of an MPI communication call that is passed a
29
           communication buffer with type defined by a derived datatype (Section 5.1). If this
30
           communicator buffer contains variables of type CHARACTER then the information on
31
           their length will not be passed to the MPI routine.
32
           This problem forces us to provide explicit information on character length with the
33
           MPI call. One could add a length parameter to the type MPI_CHARACTER, but this
34
           does not add much convenience and the same functionality can be achieved by defining
35
           a suitable derived datatype. (End of rationale.)
36
37
           Advice to implementors. Some compilers pass Fortran CHARACTER arguments as a
38
           structure with a length and a pointer to the actual string. In such an environment,
39
           the MPI call needs to dereference the pointer in order to reach the string. (End of
40
           advice to implementors.)
41
42
     3.3.2
            Data Conversion
43
44
      One of the goals of MPI is to support parallel computations across heterogeneous environ-
45
      ments. Communication in a heterogeneous environment may require data conversions. We
46
      use the following terminology.
47
      type conversion changes the datatype of a value, e.g., by rounding a REAL to an INTEGER.
48
```

representation conversion changes the binary representation of a value, e.g., from Hex floating point to IEEE floating point.

The type matching rules imply that MPI communication never entails type conversion. On the other hand, MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical and character values, and to convert a floating point value to the nearest value that can be represented on the target system.

Overflow and underflow exceptions may occur during floating point conversions. Conversion of integers or characters may also lead to exceptions when a value that can be represented in one system cannot be represented in the other system. An exception occurring during representation conversion results in a failure of the communication. An error occurs either in the send operation, or the receive operation, or both.

If a value sent in a message is untyped (i.e., of type MPI_BYTE), then the binary representation of the byte stored at the receiver is identical to the binary representation of the byte loaded at the sender. This holds true, whether sender and receiver run in the same or in distinct environments. No representation conversion is required. (Note that representation conversion may occur when values of type MPI_CHARACTER or MPI_CHAR are transferred, for example, from an EBCDIC encoding to an ASCII encoding.)

No conversion need occur when an MPI program executes in a homogeneous system, where all processes run in the same environment.

Consider the three examples, 3.1-3.3. The first program is correct, assuming that **a** and **b** are REAL arrays of size ≥ 10 . If the sender and receiver execute in different environments, then the ten real values that are fetched from the send buffer will be converted to the representation for reals on the receiver site before they are stored in the receive buffer. While the number of real elements fetched from the send buffer equal the number of real elements stored in the receive buffer, the number of bytes stored need not equal the number of bytes loaded. For example, the sender may use a four byte representation and the receiver an eight byte representation for reals.

The second program is erroneous, and its behavior is undefined.

The third program is correct. The exact same sequence of forty bytes that were loaded from the send buffer will be stored in the receive buffer, even if sender and receiver run in a different environment. The message sent has exactly the same length (in bytes) and the same binary representation as the message received. If **a** and **b** are of different types, or if they are of the same type but different data representations are used, then the bits stored in the receive buffer may encode values that are different from the values they encoded in the send buffer.

Data representation conversion also applies to the envelope of a message: source, destination and tag are all integers that may need to be converted.

Advice to implementors. The current definition does not require messages to carry data type information. Both sender and receiver provide complete data type information. In a heterogeneous environment, one can either use a machine independent encoding such as XDR, or have the receiver convert from the sender representation to its own, or even have the sender do the conversion.

Additional type information might be added to messages in order to allow the system to detect mismatches between datatype at sender and receiver. This might be 47 particularly useful in a slower but safer debug mode. (*End of advice to implementors.*) 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

 31

32

33

34

35

36

37

38

39

40

41

42

43

44

MPI requires support for inter-language communication, e.g., if messages are sent using an MPI procedure from the MPI C language interface and received using an MPI procedure from one of the MPI Fortran language interfaces. The behavior is defined in Section 19.2.

3.4 Communication Modes

The send call described in Section 3.2.1 is **blocking**: it does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.

¹¹ Message buffering decouples the send and receive operations. A blocking send can com-¹² plete as soon as the message was buffered, even if no matching receive has been executed by ¹³ the receiver. On the other hand, message buffering can be expensive, as it entails additional ¹⁴ memory-to-memory copying, and it requires the allocation of memory for buffering. MPI ¹⁵ offers the choice of several communication modes that allow one to control the choice of the ¹⁶ communication protocol.

The send call described in Section 3.2.1 uses the **standard** communication mode. In this mode, it is up to MPI to decide whether outgoing messages will be buffered. MPI may buffer outgoing messages. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver.

Thus, a send in standard mode can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. The standard mode send is *non-local*: successful completion of the send operation may depend on the occurrence of a matching receive.

Rationale. The reluctance of MPI to mandate whether standard sends are buffering or not stems from the desire to achieve portable programs. Since any system will run out of buffer resources as message sizes are increased, and some implementations may want to provide little buffering, MPI takes the position that correct (and therefore, portable) programs do not rely on system buffering in standard mode. Buffering may improve the performance of a correct program, but it doesn't affect the result of the program. If the user wishes to guarantee a certain amount of buffering, the userprovided buffer system of Section 3.6 should be used, along with the buffered-mode send. (*End of rationale.*)

³⁹ There are three additional communication modes.

40 A **buffered** mode send operation can be started whether or not a matching receive 41 has been posted. It may complete before a matching receive is posted. However, unlike the 42standard send, this operation is *local*, and its completion does not depend on the occurrence 43of a matching receive. Thus, if a send is executed and no matching receive is posted, then 44MPI must buffer the outgoing message, so as to allow the send call to complete. An error will 45occur if there is insufficient buffer space. The amount of available buffer space is controlled 46by the user — see Section 3.6. Buffer allocation by the user may be required for the buffered 47mode to be effective.

48

1

 $\mathbf{2}$

3

 $\frac{4}{5}$

6 7

8

9

10

29

30

 31

32

33

34

35

36

37

According to the definitions in Section 2.4.2, MPI_BSEND is a completing procedure and the user can re-use all resources given as arguments, including the message data buffer. It is also a local procedure because it returns immediately without depending on the execution of any MPI procedure in any other MPI process.

Advice to users. This is one of the exceptions in which a completing and therefore blocking operation-related procedure is local. (*End of advice to users.*)

A send that uses the **synchronous** mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but it also indicates that the receiver has reached a certain point in its execution, namely that it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is *non-local*.

A send that uses the **ready** communication mode may be started *only* if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined. On some systems, this allows the removal of a hand-shake operation that is otherwise required and results in improved performance. The completion of the send operation does not depend on the status of a matching receive, and merely indicates that the send buffer can be reused. A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation; it is merely that the sender provides additional information to the system (namely that a matching receive is already posted), that can save some overhead. In a correct program, therefore, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance.

Three additional send functions are provided for the three additional communication modes. The communication mode is indicated by a one letter prefix: B for buffered, S for synchronous, and R for ready.

	_			
- J	N	buf	initial address of send buffer (choice)	34
, i	N	count	number of elements in send buffer (non-negative	35
I	N	count		36
			integer)	37
I	N	datatype	datatype of each send buffer element (handle)	38
П	N	dest	rank of destination (integer)	39
				40
I	N	tag	message tag (integer)	41
I	N	comm	communicator (handle)	42
				43

MPI_BSEND(buf, count, datatype, dest, tag, comm)

C binding

Fortran 2008 binding

 24

2 3 4 5 6 7 8	TYPE(INTEG TYPE() TYPE() INTEG Fortran b MPI_BSEND <type INTEG</type 	<pre>(buf, count, datatype, de *), DIMENSION(), INTENT ER, INTENT(IN) :: count, MPI_Datatype), INTENT(IN) MPI_Comm), INTENT(IN) :: ER, OPTIONAL, INTENT(OUT) inding (BUF, COUNT, DATATYPE, DE > BUF(*) ER COUNT, DATATYPE, DEST, n buffered mode.</pre>	C(IN) :: buf dest, tag :: datatype comm :: ierror EST, TAG, COMM, IERROR)
14 15	MPI_SSEN	D(buf, count, datatype, dest, t	ag, comm)
16	IN	buf	initial address of send buffer (choice)
17 18 19	IN	count	number of elements in send buffer (non-negative integer)
20	IN	datatype	datatype of each send buffer element (handle)
21	IN	dest	rank of destination (integer)
22 23	IN	tag	message tag (integer)
23 24	IN	comm	communicator (handle)
27 28 29 30 31 1 32 33 34 35 36 37	Fortran 2 MPI_Ssend TYPE(INTEG TYPE() TYPE() INTEG Fortran b MPI_SSEND <type INTEG</type 	<pre>send(const void *buf, int</pre>	est, tag, comm, ierror) C(IN) :: buf dest, tag :: datatype comm :: ierror EST, TAG, COMM, IERROR)

buf	initial address of send buffer (choice)	2
count	number of elements in send buffer (non-negative integer)	4
datatype	datatype of each send buffer element (handle)	6
dest	rank of destination (integer)	5
tag	message tag (integer)	ç
comm	communicator (handle)	1
	count datatype dest tag	countnumber of elements in send buffer (non-negative integer)datatypedatatype of each send buffer element (handle)destrank of destination (integer)tagmessage tag (integer)

C binding

Fortran 2008 binding

MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)	
TYPE(*), DIMENSION(), INTENT(IN) :: buf	
INTEGER, INTENT(IN) :: count, dest, tag	
TYPE(MPI_Datatype), INTENT(IN) :: datatype	
TYPE(MPI_Comm), INTENT(IN) :: comm	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	

Fortran binding

MPI_RSEND(BUF,	COUNT, DATATY	YPE, DEST, T	AG, COMM,	IERROR)
<type> BUF</type>	(*)			
INTEGER CO	UNT, DATATYPE	, DEST, TAG,	COMM, IER	ROR

Send in ready mode.

There is only one receive operation, but it matches any of the send modes. The receive operation described in the last section is *blocking*: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started).

In a multithreaded implementation of MPI, the system may de-schedule a thread that is blocked on a send or receive operation, and schedule another thread for execution in the same address space. In such a case it is the user's responsibility not to modify a communication buffer until the communication completes. Otherwise, the outcome of the computation is undefined.

Advice to implementors. Since a synchronous send cannot complete before a matching receive is posted, one will not normally buffer messages sent by such an operation.

It is recommended to choose buffering over blocking the sender, whenever possible, for standard sends. The programmer can signal his or her preference for blocking the sender until a matching receive occurs by using the synchronous send mode.

A possible communication protocol for the various communication modes is outlined below.

ready send: The message is sent as soon as possible.

 31

synchronous send: The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message, and the sender now sends the message.

- standard send: First protocol may be used for short messages, and second protocol for long messages.
- *buffered send*: The sender copies the message into a buffer and then sends it with a nonblocking send (using the same protocol as for standard send).
- Additional control messages might be needed for flow control and error recovery. Of
 course, there are many other possible protocols.
- Ready send can be implemented as a standard send. In this case there will be no performance advantage (or disadvantage) for the use of ready send.
- A standard send can be implemented as a synchronous send. In such a case, no data buffering is needed. However, users may expect some buffering.
 - In a multithreaded environment, the execution of a blocking communication should block only the executing thread, allowing the thread scheduler to de-schedule this thread and schedule another thread for execution. (*End of advice to implementors.*)
- 18 19 20

21 22

23

24

16

17

3.5 Semantics of Point-to-Point Communication

A valid MPI implementation guarantees certain general properties of point-to-point communication, which are described in this section.

25Order Messages are *non-overtaking*: If a sender sends two messages in succession to the 26same destination, and both match the same receive, then this operation cannot receive the 27second message if the first one is still pending. If a receiver posts two receives in succession, 28and both match the same message, then the second receive operation cannot be satisfied 29 by this message, if the first one is still pending. This requirement facilitates matching of 30 sends to receives. It guarantees that message-passing code is deterministic, if processes are 31 single-threaded and the wildcard MPI_ANY_SOURCE is not used in receives. (Some of the 32 calls described later, such as MPI_CANCEL or MPI_WAITANY, are additional sources of 33 nondeterminism.) 34

If a process has a single thread of execution, then any two communications executed 35 by this process are ordered. On the other hand, if the process is multithreaded, then the 36 semantics of thread execution may not define a relative order between two send operations 37 executed by two distinct threads. The operations are logically concurrent, even if one 38 physically precedes the other. In such a case, the two messages sent can be received in 39 any order. Similarly, if two receive operations that are logically concurrent receive two 40 successively sent messages, then the two messages can match the two receives in either 41 order. 42

⁴³ **Example 3.5** An example of non-overtaking messages.

- 45
- 46
- 40 47
- 48

1

2

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

The message sent by the first send must be received by the first receive, and the message sent by the second send must be received by the second receive.

Progress If a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independently of other actions in the system: the send operation will complete, unless the receive is satisfied by another message, and completes; the receive operation will complete, unless the message sent is consumed by another matching receive that was posted at the same destination process.

Example 3.6 An example of two, intertwined matching pairs.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
CALL MPI_SSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

Both processes invoke their first communication call. Since the first send of process zero uses the buffered mode, it must complete, irrespective of the state of process one. Since no matching receive is posted, the message will be copied into buffer space. (If insufficient buffer space is available, then the program will fail.) The second send is then invoked. At that point, a matching pair of send and receive operation is enabled, and both operations must complete. Process one next invokes its second receive call, which will be satisfied by the buffered message. Note that process one received the messages in the reverse order they were sent.

Fairness MPI makes no guarantee of *fairness* in the handling of communication. Suppose that a send is posted. Then it is possible that the destination process repeatedly posts a receive that matches this send, yet the message is never received, because it is each time overtaken by another message, sent from another source. Similarly, suppose that a receive was posted by a multithreaded process. Then it is possible that messages that match this receive are repeatedly received, yet the receive is never satisfied, because it is overtaken by other receives posted at this node (by other executing threads). It is the programmer's responsibility to prevent starvation in such situations.

 $46 \\ 47$

 31

1 **Resource** limitations Any pending communication operation consumes system resources $\mathbf{2}$ that are limited. Errors may occur when lack of resources prevent the execution of an MPI 3 call. A quality implementation will use a (small) fixed amount of resources for each pending 4 send in the ready or synchronous mode and for each pending receive. However, buffer space $\mathbf{5}$ may be consumed to store messages sent in standard mode, and must be consumed to store 6 messages sent in buffered mode, when no matching receive is available. The amount of space 7available for buffering will be much smaller than program data memory on many systems. 8 Then, it will be easy to write programs that overrun available buffer space.

⁹ MPI allows the user to provide buffer memory for messages sent in the buffered mode. ¹⁰ Furthermore, MPI specifies a detailed operational model for the use of this buffer. An MPI ¹¹ implementation is required to do no worse than implied by this model. This allows users to ¹² avoid buffer overflows when they use buffered sends. Buffer allocation and use is described ¹³ in Section 3.6.

14A buffered send operation that cannot complete because of a lack of buffer space is 15erroneous. When such a situation is detected, an error is signaled that may cause the 16program to terminate abnormally. On the other hand, a standard send operation that 17cannot complete because of lack of buffer space will merely block, waiting for buffer space 18 to become available or for a matching receive to be posted. This behavior is preferable in 19many situations. Consider a situation where a producer repeatedly produces new values 20and sends them to a consumer. Assume that the producer produces new values faster 21than the consumer can consume them. If buffered sends are used, then a buffer overflow 22will result. Additional synchronization has to be added to the program so as to prevent 23this from occurring. If standard sends are used, then the producer will be automatically 24 throttled, as its send operations will block when buffer space is unavailable.

In some situations, a lack of buffer space leads to deadlock situations. This is illustrated
 by the examples below.

```
Example 3.7 An exchange of messages.
```

```
29
     CALL MPI_COMM_RANK(comm, rank, ierr)
30
     IF (rank .EQ. 0) THEN
^{31}
        CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
32
        CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
33
     ELSE IF (rank .EQ. 1) THEN
34
       CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
35
        CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
36
     END IF
37
38
```

```
This program will succeed even if no buffer space for data is available. The standard send operation can be replaced, in this example, with a synchronous send.
```

```
    Example 3.8 An errant attempt to exchange messages.
    CALL MPI_COMM_RANK(comm, rank, ierr)
    IF (rank .EQ. 0) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    ELSE IF (rank .EQ. 1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
```

27

CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr) END IF

The receive operation of the first process must complete before its send, and can complete only if the matching send of the second processor is executed. The receive operation of the second process must complete before its send and can complete only if the matching send of the first process is executed. This program will always deadlock. The same holds for any other send mode.

Example 3.9 An exchange that relies on buffering.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

The message sent by each process has to be copied out before the send operation returns and the receive operation starts. For the program to complete, it is necessary that at least one of the two messages sent be buffered. Thus, this program can succeed only if the communication system can buffer at least **count** words of data.

Advice to users. When standard send operations are used, then a deadlock situation may occur where both processes are blocked because buffer space is not available. The same will certainly happen, if the synchronous mode is used. If the buffered mode is used, and not enough buffer space is available, then the program will not complete either. However, rather than a deadlock situation, we shall have a buffer overflow error.

A program is "safe" if no message buffering is required for the program to complete. One can replace all sends in such program with synchronous sends, and the program will still run correctly. This conservative programming style provides the best portability, since program completion does not depend on the amount of buffer space available or on the communication protocol used.

Many programmers prefer to have more leeway and opt to use the "unsafe" programming style shown in Example 3.9. In such cases, the use of standard sends is likely to provide the best compromise between performance and robustness: quality implementations will provide sufficient buffering so that "common practice" programs will not deadlock. The buffered send mode can be used for programs that require more buffering, or in situations where the programmer wants more control. This mode might also be used for debugging purposes, as buffer overflow conditions are easier to diagnose than deadlock conditions.

Nonblocking message-passing operations, as described in Section 3.7, can be used to avoid the need for buffering outgoing messages. This prevents deadlocks due to lack of buffer space, and improves performance, by allowing overlap of computation and communication, and avoiding the overheads of allocating buffers and copying messages into buffers. (*End of advice to users.*) 44 45 46 47 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

7

8 9

10

11

12

13

14

15

16

17

18 19

20

21

22

23 24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

```
Buffer Allocation and Usage
1
     3.6
\mathbf{2}
     A user may specify a buffer to be used for buffering messages sent in buffered mode. Buffer-
3
     ing is done by the sender.
4
5
6
     MPI_BUFFER_ATTACH(buffer, size)
7
       IN
                 buffer
                                              initial buffer address (choice)
8
9
       IN
                 size
                                              buffer size, in bytes (non-negative integer)
10
11
     C binding
12
     int MPI_Buffer_attach(void *buffer, int size)
13
14
     Fortran 2008 binding
     MPI_Buffer_attach(buffer, size, ierror)
15
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
16
          INTEGER, INTENT(IN) :: size
17
18
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
     Fortran binding
20
     MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
21
          <type> BUFFER(*)
22
          INTEGER SIZE, IERROR
23
^{24}
          Provides to MPI a buffer in the user's memory to be used for buffering outgoing mes-
25
     sages. The buffer is used only by messages sent in buffered mode. Only one buffer can be
26
     attached to a process at a time. In C, buffer is the starting address of a memory region. In
27
     Fortran, one can pass the first element of a memory region or a whole array, which must be
     'simply contiguous' (for 'simply contiguous,' see also Section 19.1.12).
28
29
30
     MPI_BUFFER_DETACH(buffer_addr, size)
^{31}
32
       OUT
                 buffer_addr
                                              initial buffer address (choice)
33
       OUT
                                              buffer size, in bytes (non-negative integer)
                 size
34
35
     C binding
36
     int MPI_Buffer_detach(void *buffer_addr, int *size)
37
38
     Fortran 2008 binding
39
     MPI_Buffer_detach(buffer_addr, size, ierror)
40
          USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
41
          TYPE(C_PTR), INTENT(OUT) :: buffer_addr
42
          INTEGER, INTENT(OUT) :: size
43
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     Fortran binding
45
     MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
46
          <type> BUFFER_ADDR(*)
47
          INTEGER SIZE, IERROR
48
```

Detach the buffer currently associated with MPI. The call returns the address and the size of the detached buffer. This operation will block until all messages currently in the buffer have been transmitted. Upon return of this function, the user may reuse or deallocate the space taken by the buffer.

Example 3.10 Calls to attach and detach buffers.

```
#define BUFFSIZE 10000
int size;
char *buff;
MPI_Buffer_attach(malloc(BUFFSIZE), BUFFSIZE);
/* a buffer of 10000 bytes can now be used by MPI_Bsend */
MPI_Buffer_detach(&buff, &size);
/* Buffer size reduced to zero */
MPI_Buffer_attach(buff, size);
/* Buffer of 10000 bytes available again */
```

Advice to users. Even though the C functions MPI_Buffer_attach and MPI_Buffer_detach both have a first argument of type void*, these arguments are used differently: A pointer to the buffer is passed to MPI_Buffer_attach; the address of the pointer is passed to MPI_Buffer_detach, so that this call can return the pointer value. In Fortran with the mpi module or mpif.h, the type of the buffer_addr argument is wrongly defined and the argument is therefore unused. In Fortran with the mpi_f08 module, the address of the buffer is returned as TYPE(C_PTR), see also Example 9.1 about the use of C_PTR pointers. (End of advice to users.)

Rationale. Both arguments are defined to be of type void* (rather than void* and void**, respectively), so as to avoid complex type casts. E.g., in the last example, &buff, which is of type char**, can be passed as argument to MPI_Buffer_detach without type casting. If the formal parameter had type void** then we would need a type cast before and after the call. (End of rationale.)

The statements made in this section describe the behavior of MPI for buffered-mode sends. When no buffer is currently associated, MPI behaves as if a zero-sized buffer is associated with the process.

MPI must provide as much buffering for outgoing messages *as if* outgoing message data were buffered by the sending process, in the specified buffer space, using a circular, contiguous-space allocation policy. We outline below a model implementation that defines this policy. MPI may provide more buffering, and may use a better buffer allocation algorithm than described below. On the other hand, MPI may signal an error whenever the simple buffering allocator described below would run out of space. In particular, if no buffer is explicitly associated with the process, then any buffered send may cause an error.

MPI does not provide mechanisms for querying or controlling buffering done by standard mode sends. It is expected that vendors will provide such information for their implementations.

Rationale. There is a wide spectrum of possible implementations of buffered communication: buffering can be done at sender, at receiver, or both; buffers can be dedicated to one sender-receiver pair, or be shared by all communications; buffering

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

 $45 \\ 46$

can be done in real or in virtual memory; it can use dedicated memory, or memory shared by other processes; buffer space may be allocated statically or be changed dynamically; etc. It does not seem feasible to provide a portable mechanism for querying or controlling buffering that would be compatible with all these choices, yet provide meaningful information. (*End of rationale.*)

3.6.1 Model Implementation of Buffered Mode

The model implementation uses the packing and unpacking functions described in Section 5.2 and the nonblocking communication functions described in Section 3.7.

We assume that a circular queue of pending message entries (PME) is maintained. Each entry contains a communication request handle that identifies a pending nonblocking send, a pointer to the next entry and the packed message data. The entries are stored in successive locations in the buffer. Free space is available between the queue tail and the queue head.

A buffered send call results in the execution of the following code.

- Traverse sequentially the PME queue from head towards the tail, deleting all entries for communications that have completed, up to the first entry with an uncompleted request; update queue head to point to that entry.
 - Compute the number, n, of bytes needed to store an entry for the new message. An upper bound on n can be computed as follows: A call to the function

MPI_PACK_SIZE(count, datatype, comm, size), with the count, datatype and comm arguments used in the MPI_BSEND call, returns an upper bound on the amount of space needed to buffer the message data (see Section 5.2). The MPI constant MPI_BSEND_OVERHEAD provides an upper bound on the additional space consumed by the entry (e.g., for pointers or envelope information).

- Find the next contiguous empty space of n bytes in buffer (space following queue tail, or space at start of buffer if queue tail is too close to end of buffer). If space is not found then raise buffer overflow error.
 - Append to end of PME queue in contiguous space the new entry that contains request handle, next pointer and packed message data; MPI_PACK is used to pack data.
 - Post nonblocking send (standard mode) for packed data.
 - Return

3.7 Nonblocking Communication

41 **Nonblocking communication** is important both for reasons of correctness and perfor-42mance. For complex communication patterns, the use of only blocking communication 43 (without buffering) is difficult because the programmer must ensure that each send is 44matched with a receive in an order that avoids *deadlock*. For communication patterns that 45are determined only at run time, this is even more difficult. Nonblocking communication 46can be used to avoid this problem, allowing programmers to express complex and possibly 47dynamic communication patterns without needing to ensure that all sends and receives 48 are issued in an order that prevents deadlock (see Section 3.5 and the discussion of "safe"

Unofficial Draft for Comment Only

1

2

3

4

5

6 7

8

9

10

16 17

18

19

20 21

22

23

 24

25

26

27

28

29

30

31 32

33

34 35

36

37 38 39

programs). Nonblocking communication also allows for the overlap of communication with different communication operations, e.g., to prevent the *serialization* of such operations, and for the *overlap* of communication with computation. Whether an implementation is able to accomplish an effective (from a performance standpoint) overlap of operations depends on the implementation itself and the system on which the implementation is running. Using nonblocking operations *permits* an implementation to overlap communication with computation, but does not require it to do so.

A nonblocking **send start** call initiates the send operation, but does not complete it. 8 9 The send start call can return before the message was copied out of the send buffer. A 10 separate send complete call is needed to complete the communication, i.e., to verify that 11the data has been copied out of the send buffer. With suitable hardware, the transfer of data 12out of the sender memory may proceed concurrently with computations done at the sender 13 after the send was initiated and before it completed. Similarly, a nonblocking receive start call initiates the receive operation, but does not complete it. The call can return before a 1415message is stored into the receive buffer. A separate **receive complete** call is needed to 16complete the receive operation and verify that the data has been received into the receive 17 buffer. With suitable hardware, the transfer of data into the receiver memory may proceed 18 concurrently with computations done after the receive was initiated and before it completed. 19The use of nonblocking receives may also avoid system buffering and memory-to-memory 20copying, as information is provided early on the location of the receive buffer.

Nonblocking send start calls can use the same four modes as blocking sends: *standard*, *buffered*, *synchronous* and *ready*. These carry the same meaning. Sends of all modes, *ready* excepted, can be started whether a matching receive has been posted or not; a nonblocking **ready** send can be started only if a matching receive is posted. In all cases, the send start call is local: it returns immediately, irrespective of the status of other processes. If the call causes some system resource to be exhausted, then it will fail and return an error code. Quality implementations of MPI should ensure that this happens only in "pathological" cases. That is, an MPI implementation should be able to support a large number of pending nonblocking operations.

The send-complete call returns when data has been copied out of the send buffer. It may carry additional meaning, depending on the send mode.

If the send mode is **synchronous**, then the send can complete only if a matching receive has started. That is, a receive has been posted, and has been matched with the send. In this case, the send-complete call is non-local. Note that a synchronous, nonblocking send may complete, if matched by a nonblocking receive, before the receive complete call occurs. (It can complete as soon as the sender "knows" the transfer will complete, but before the receiver "knows" the transfer will complete.)

If the send mode is **buffered** then the message must be buffered if there is no pending receive. In this case, the send-complete call is local, and must succeed irrespective of the status of a matching receive.

If the send mode is **standard** then the send-complete call may return before a matching receive is posted, if the message is buffered. On the other hand, the receive-complete may not complete until a matching receive is posted, and the message was copied into the receive buffer.

Nonblocking sends can be matched with blocking receives, and vice-versa.

Advice to users. The completion of a send operation may be delayed, for standard mode, and must be delayed, for synchronous mode, until a matching receive is posted.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

 $\overline{7}$

21

22

23 24

25 26

27 28

29

30

31

32 33

34

35

36 37

38 39

40 41

42

43 44

 $45 \\ 46$

47

The use of nonblocking sends in these two cases allows the sender to proceed ahead of the receiver, so that the computation is more tolerant of fluctuations in the speeds of the two processes.

Nonblocking sends in the buffered and ready modes have a more limited impact, e.g., the blocking version of buffered send is capable of completing regardless of when a matching receive call is made. However, separating the start from the completion of these sends still gives some opportunity for optimization within the MPI library. For example, starting a buffered send gives an implementation more flexibility in determining if and how the message is buffered. There are also advantages for both nonblocking buffered and ready modes when data copying can be done concurrently with computation.

The message-passing model implies that communication is initiated by the sender. The communication will generally have lower overhead if a receive is already posted when the sender initiates the communication (data can be moved directly to the receive buffer, and there is no need to queue a pending send request). However, a receive operation can complete only after the matching send has occurred. The use of nonblocking receives allows one to achieve lower communication overheads without blocking the receiver while it waits for the send. (*End of advice to users.*)

19 20 21

3.7.1 Communication Request Objects

Nonblocking communications use opaque **request** objects to identify communication operations and match the operation that initiates the communication with the operation that terminates it. These are system objects that are accessed via a handle. A request object identifies various properties of a communication operation, such as the send mode, the communication buffer that is associated with it, its context, the tag and destination arguments to be used for a send, or the tag and source arguments to be used for a receive. In addition, this object stores information about the status of the pending communication operation.

29 30

 31

3.7.2 Communication Initiation

For the functions defined in this section, we use the same naming conventions as for blocking
 communication: a prefix of B, S, or R is used for buffered, synchronous or ready mode.
 In addition, for these functions a prefix of I (for immediate and incomplete) indicates
 that the call is nonblocking.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

3.7. NONBLOCKING COMMUNICATION

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)			
IN	buf	initial address of send buffer (choice)	2
IN	count	number of elements in send buffer (non-negative integer)	3 4 5
IN	datatype	datatype of each send buffer element (handle)	6
IN	dest	rank of destination (integer)	7
IN	tag	message tag (integer)	8 9
IN	comm	communicator (handle)	10
OUT	request	communication request (handle)	11
			12 13

C binding

C binding	
<pre>int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	
<pre>int tag, MPI_Comm comm, MPI_Request *request)</pre>	
Fortran 2008 binding	

Fortran 2008 binding

MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf
INTEGER, INTENT(IN) :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding

Fortran binding

MPI_ISEND(BUF,	COUNT, DATATYF	PE, DEST,	TAG, COMM,	REQUEST,	IERROR)
<type> BUF(</type>	(*)				
INTEGER COU	INT, DATATYPE,	DEST, TAG	, COMM, RE	QUEST, IER	ROR
~ .					

Start a standard mode, nonblocking send.

MPI_IBSEND(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)	34
IN	count	number of elements in send buffer (non-negative integer)	35 36
IN	datatype	datatype of each send buffer element (handle)	37 38
IN	dest	rank of destination (integer)	39 40
IN	tag	message tag (integer)	40 41
IN	comm	communicator (handle)	42
OUT	request	communication request (handle)	43
		- 、 ,	44
			45

C binding

int MPI_Ibsend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror)
3
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
4
          INTEGER, INTENT(IN) :: count, dest, tag
5
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         TYPE(MPI_Comm), INTENT(IN) :: comm
7
         TYPE(MPI_Request), INTENT(OUT) :: request
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     Fortran binding
10
     MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
11
          <type> BUF(*)
12
          INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
13
14
         Start a buffered mode, nonblocking send.
15
16
     MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)
17
18
                                            initial address of send buffer (choice)
       IN
                 buf
19
       IN
                                            number of elements in send buffer (non-negative
                count
20
                                            integer)
21
       IN
                datatype
                                            datatype of each send buffer element (handle)
22
23
       IN
                dest
                                            rank of destination (integer)
24
       IN
                tag
                                            message tag (integer)
25
       IN
                comm
                                            communicator (handle)
26
27
       OUT
                                            communication request (handle)
                request
28
29
     C binding
30
     int MPI_Issend(const void *buf, int count, MPI_Datatype datatype, int dest,
^{31}
                    int tag, MPI_Comm comm, MPI_Request *request)
32
     Fortran 2008 binding
33
34
     MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
35
         INTEGER, INTENT(IN) :: count, dest, tag
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         TYPE(MPI_Request), INTENT(OUT) :: request
39
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     Fortran binding
42
     MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
43
          <type> BUF(*)
44
          INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
45
         Start a synchronous mode, nonblocking send.
46
47
48
```

IN buf initial address of send buffer (choice) IN count number of elements in send buffer (non-negative integer) IN datatype datatype of each send buffer element (handle) IN dest rank of destination (integer) IN dest rank of destination (integer) IN tag message tag (integer) IN comm communicator (handle) OUT request communicator (handle) OUT request communicator (handle) OUT request communicator (handle) Cbinding fint tag, MPI_Comm comm, MPI_Batatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request +request) Fortran 2008 binding Fill_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) TYPE(MPI_Request), INTENT(IN) :: count, dest, tag fintEGER, INTERT(IN) :: count TYPE(MPI_Comm), INTENT(OUT) :: request intraGeR, OPTIONAL, INTENT(OUT) :: request TYPE(MPI_Request), INTENT(IN) :: count fintEGER Count, Datatype, DEST, TAG, COMM, REQUEST, IERROR) ctype> EUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR mumber of elements in receive buffer (choice) IN	MPI_IRSI	END(buf, count, datat	ype, dest, tag, comm, request)	
integer) IN datatype datatype of each send buffer element (handle) IN dest rank of destination (integer) IN tag message tag (integer) IN comm communicator (handle) OUT request communication request (handle) OUT int tag, MPI_Comm comm, MPI_Request *request) int dest, int tag, MPI_Comm comm, MPI_Request *request) Fortran 2008 binding INTENT(IN) :: count, dest, tag trype(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Datatype), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(OUT) :: ierror TYPE(MPI_Request), INTENT(OUT) :: ierror Fortran binding INTESEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) cype> EUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. MPI_IRECV(buf, count, datatype, source, tag, comm, request) OUT buf OUT buf initial address of receive buffer (choice) int	IN	buf	initial address of send buffer (choice)	
IN dest rank of destination (integer) IN tag message tag (integer) IN comm communicator (handle) OUT request communication request (handle) C binding	IN	count	· · · · · ·	
IN tag message tag (integer) IN comm communicator (handle) OUT request communication request (handle) C binding Int MPI_Irsend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request) Fortran 2008 binding (PI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :; buf INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Comm), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding (PI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. MPI_IRECV(buf, count, datatype, source, tag, comm, request) OUT buf initial address of receive buffer (choice) IN count number of elements in receive buffer (non-negative integer) IN datatype datatype of each receive buffer element (handle) IN source rank of source or MPI_ANY_SOURCE (integer) IN tag message tag or MPI_ANY_TAG (integer) IN comm communicator (handle)</type>	IN	datatype	datatype of each send buffer element (handle)	
IN comm communicator (handle) OUT request communication request (handle) C binding int tag, MPI_Comm comm, MPI_Batatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request) Fortran 2008 binding [PI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(IN) :: datatype TYPE(MPI_Request), INTENT(IN) :: domm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding PPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <tp><tp><tp><tp>INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. MPI_IRECV(buf, count, datatype, source, tag, comm, request) QUT buf number of elements in receive buffer (choice) IN count number of elements in receive buffer (non-negative integer) IN datatype IN datatype IN count</tp></tp></tp></tp>	IN	dest	rank of destination (integer)	
IN comm communicator (handle) OUT request communication request (handle) C binding int tag, MPI_Comm comm, MPI_Batatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request) Fortran 2008 binding [PI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(IN) :: datatype TYPE(MPI_Request), INTENT(IN) :: detatype TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding PPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <tp><tp><tp><tp>INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. MPI_IRECV(buf, count, datatype, source, tag, comm, request) QUT buf number of elements in receive buffer (choice) IN count number of elements in receive buffer (non-negative integer) IN IN datatype IN count</tp></tp></tp></tp>	IN	tag	message tag (integer)	
OUT request communication request (handle) C binding Int MPI_Irsend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request) Fortran 2008 binding [PPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding PPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. MPI_IRECV(buf, count, datatype, source, tag, comm, request) QUT buf initial address of receive buffer (choice) IN count number of elements in receive buffer (non-negative integer) IN datatype datatype of each receive buffer element (handle) IN source rank of source or MPI_ANY_SOURCE (integer) IN tag message tag or MPI_ANY_TAG (integer) IN comm communicator (handle)</type>	IN	-		
C binding Int MPI_Irsend(const void *buf, int count, MPI_Datatype datatype, int dest,				
Fortran 2008 binding IPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding PPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. MPI_IRECV(buf, count, datatype, source, tag, comm, request) QUT buf initial address of receive buffer (choice) IN count number of elements in receive buffer (non-negative integer) IN datatype IN datatype IN adatype IN adatype IN tag IN message tag or MPI_ANY_SOURCE (integer) IN comm</type>		_Irsend(const void		
MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(UUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. MPI_IRECV(buf, count, datatype, source, tag, comm, request) OUT buf initial address of receive buffer (choice) N number of elements in receive buffer (non-negative integer) IN datatype IN datatype IN source rank of source or MPI_ANY_SOURCE (integer) IN tag message tag or MPI_ANY_TAG (integer) IN comm</type>	- .	C		
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding PPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. MPI_IRECV(buf, count, datatype, source, tag, comm, request) OUT buf initial address of receive buffer (choice) N count number of elements in receive buffer (non-negative integer) IN datatype IN datatype IN source rank of source or MPI_ANY_SOURCE (integer) IN tag message tag or MPI_ANY_TAG (integer) IN comm</type>		0	sature dest tag comm request jerror)	
INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding PPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. MPI_IRECV(buf, count, datatype, source, tag, comm, request) OUT buf initial address of receive buffer (choice) IN count number of elements in receive buffer (non-negative integer) IN datatype datatype of each receive buffer element (handle) IN source rank of source or MPI_ANY_SOURCE (integer) IN tag message tag or MPI_ANY_TAG (integer) IN comm communicator (handle)</type>				
TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding IPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. MPI_IRECV(buf, count, datatype, source, tag, comm, request) OUT buf initial address of receive buffer (choice) IN count number of elements in receive buffer (non-negative integer) IN datatype IN source IN source IN tag IN tag IN comm Comm comm</type>				
TYPE (MPI_Request), INTENT (OUT) :: request INTEGER, OPTIONAL, INTENT (OUT) :: ierror Fortran binding PPI_IRSEND (BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. MPI_IRECV(buf, count, datatype, source, tag, comm, request) OUT buf initial address of receive buffer (choice) IN count number of elements in receive buffer (non-negative integer) IN datatype IN datatype IN source IN rank of source or MPI_ANY_SOURCE (integer) IN tag IN comm</type>	TYPE	E(MPI_Datatype), II	NTENT(IN) :: datatype	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror Yortran binding PI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. MPI_IRECV(buf, count, datatype, source, tag, comm, request) OUT buf initial address of receive buffer (choice) IN count number of elements in receive buffer (non-negative integer) IN datatype IN datatype of each receive buffer (handle) IN source IN tag IN tag IN comm</type>	1111		51	
Fortran binding PI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. API_IRECV(buf, count, datatype, source, tag, comm, request) OUT buf initial address of receive buffer (choice) IN count number of elements in receive buffer (non-negative integer) IN datatype IN datatype IN source IN source IN comm IN comm</type>	TYPE	E(MPI_Comm), INTEN	Γ(IN) :: comm	
PI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. API_IRECV(buf, count, datatype, source, tag, comm, request) OUT buf initial address of receive buffer (choice) IN count number of elements in receive buffer (non-negative integer) IN datatype IN source IN source or MPI_ANY_SOURCE (integer) IN tag IN comm comm communicator (handle)</type>	TYPE TYPE	E(MPI_Comm), INTEN E(MPI_Request), IN	T(IN) :: comm TENT(OUT) :: request	
<type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERRORStart a ready mode nonblocking send.VPI_IRECV(buf, count, datatype, source, tag, comm, request)OUTbufinitial address of receive buffer (choice)INcountatatypedatatype of elements in receive buffer (non-negative integer)INdatatypeINsourceINsourceINtagINtagINcommCommcommunicator (handle)</type>	TYPE TYPE	E(MPI_Comm), INTEN E(MPI_Request), IN	T(IN) :: comm TENT(OUT) :: request	
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERRORStart a ready mode nonblocking send.API_IRECV(buf, count, datatype, source, tag, comm, request)OUTbufinitial address of receive buffer (choice)INcountnumber of elements in receive buffer (non-negative integer)INdatatypedatatype of each receive buffer element (handle)INsourcerank of source or MPI_ANY_SOURCE (integer)INtagmessage tag or MPI_ANY_TAG (integer)INcommcommunicator (handle)	TYPE TYPE INTE	E(MPI_Comm), INTEN E(MPI_Request), IN EGER, OPTIONAL, IN	T(IN) :: comm TENT(OUT) :: request	
Start a ready mode nonblocking send. API_IRECV DUT buf IN count IN count IN datatype IN datatype IN cource IN source IN tag IN tag IN tag IN comm	TYPE TYPE INTE Fortran PI_IRSE	E(MPI_Comm), INTEN E(MPI_Request), IN EGER, OPTIONAL, IN binding END(BUF, COUNT, DA	T(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror	
API_IRECV/buf, count, datatype, source, tag, comm, request)OUTbufinitial address of receive buffer (choice)INcountnumber of elements in receive buffer (non-negative integer)INdatatypedatatype of each receive buffer element (handle)INsourcerank of source or MPI_ANY_SOURCE (integer)INtagmessage tag or MPI_ANY_TAG (integer)INcommcommunicator (handle)	TYPE TYPE INTE Fortran PI_IRSE <typ< td=""><td>E(MPI_Comm), INTEN E(MPI_Request), IN EGER, OPTIONAL, IN binding END(BUF, COUNT, DA pe> BUF(*)</td><td><pre>F(IN) :: comm FENT(OUT) :: request FENT(OUT) :: ierror FATYPE, DEST, TAG, COMM, REQUEST, IERROR)</pre></td><td></td></typ<>	E(MPI_Comm), INTEN E(MPI_Request), IN EGER, OPTIONAL, IN binding END(BUF, COUNT, DA pe> BUF(*)	<pre>F(IN) :: comm FENT(OUT) :: request FENT(OUT) :: ierror FATYPE, DEST, TAG, COMM, REQUEST, IERROR)</pre>	
OUTbufinitial address of receive buffer (choice)INcountnumber of elements in receive buffer (non-negative integer)INdatatypedatatype of each receive buffer element (handle)INsourcerank of source or MPI_ANY_SOURCE (integer)INtagmessage tag or MPI_ANY_TAG (integer)INcommcommunicator (handle)	TYPE TYPE INTE Fortran PI_IRSE <typ< td=""><td>E(MPI_Comm), INTEN E(MPI_Request), IN EGER, OPTIONAL, IN binding END(BUF, COUNT, DA pe> BUF(*)</td><td><pre>F(IN) :: comm FENT(OUT) :: request FENT(OUT) :: ierror FATYPE, DEST, TAG, COMM, REQUEST, IERROR)</pre></td><td></td></typ<>	E(MPI_Comm), INTEN E(MPI_Request), IN EGER, OPTIONAL, IN binding END(BUF, COUNT, DA pe> BUF(*)	<pre>F(IN) :: comm FENT(OUT) :: request FENT(OUT) :: ierror FATYPE, DEST, TAG, COMM, REQUEST, IERROR)</pre>	
OUTbufinitial address of receive buffer (choice)INcountnumber of elements in receive buffer (non-negative integer)INdatatypedatatype of each receive buffer element (handle)INsourcerank of source or MPI_ANY_SOURCE (integer)INtagmessage tag or MPI_ANY_TAG (integer)INcommcommunicator (handle)	TYPE TYPE INTE Fortran PI_IRSE <typ INTE</typ 	E(MPI_Comm), INTEN E(MPI_Request), IN EGER, OPTIONAL, IN binding END(BUF, COUNT, DA pe> BUF(*) EGER COUNT, DATATYN	<pre>F(IN) :: comm FENT(OUT) :: request FENT(OUT) :: ierror FATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR</pre>	
OUTbufinitial address of receive buffer (choice)INcountnumber of elements in receive buffer (non-negative integer)INdatatypedatatype of each receive buffer element (handle)INsourcerank of source or MPI_ANY_SOURCE (integer)INtagmessage tag or MPI_ANY_TAG (integer)INcommcommunicator (handle)	TYPE TYPE INTE Fortran IPI_IRSE <typ INTE</typ 	E(MPI_Comm), INTEN E(MPI_Request), IN EGER, OPTIONAL, IN binding END(BUF, COUNT, DA pe> BUF(*) EGER COUNT, DATATYN	<pre>F(IN) :: comm FENT(OUT) :: request FENT(OUT) :: ierror FATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR</pre>	
INcountnumber of elements in receive buffer (non-negative integer)INdatatypedatatype of each receive buffer element (handle)INsourcerank of source or MPI_ANY_SOURCE (integer)INtagmessage tag or MPI_ANY_TAG (integer)INcommcommunicator (handle)	TYPE TYPE INTE Fortran IPI_IRSE <typ INTE Start</typ 	E(MPI_Comm), INTEN E(MPI_Request), IN EGER, OPTIONAL, IN binding END(BUF, COUNT, DA De> BUF(*) EGER COUNT, DATATYN t a ready mode nonbl	<pre>T(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror TATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR ocking send.</pre>	
INdatatypeinteger)INdatatypedatatype of each receive buffer element (handle)INsourcerank of source or MPI_ANY_SOURCE (integer)INtagmessage tag or MPI_ANY_TAG (integer)INcommcommunicator (handle)	TYPE TYPE INTE Ortran PI_IRSE <typ INTE Start</typ 	E(MPI_Comm), INTEN E(MPI_Request), IN EGER, OPTIONAL, IN binding END(BUF, COUNT, DA EGER COUNT, DATATYI t a ready mode nonbl CV(buf, count, datatyp	<pre>T(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror TATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR ocking send. be, source, tag, comm, request)</pre>	
INdatatypedatatype of each receive buffer element (handle)INsourcerank of source or MPI_ANY_SOURCE (integer)INtagmessage tag or MPI_ANY_TAG (integer)INcommcommunicator (handle)	TYPE TYPE INTE Fortran PI_IRSE <typ INTE Start MPI_IRE OUT</typ 	E(MPI_Comm), INTEN E(MPI_Request), IN EGER, OPTIONAL, IN binding END(BUF, COUNT, DA De> BUF(*) EGER COUNT, DATATYI t a ready mode nonbl CV(buf, count, datatyp buf	<pre>T(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror TATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR ocking send. be, source, tag, comm, request)</pre>	
INsourcerank of source or MPI_ANY_SOURCE (integer)INtagmessage tag or MPI_ANY_TAG (integer)INcommcommunicator (handle)	TYPE TYPE INTE Ortran PI_IRSE <typ INTE Start API_IRE OUT</typ 	E(MPI_Comm), INTEN E(MPI_Request), IN EGER, OPTIONAL, IN binding END(BUF, COUNT, DA De> BUF(*) EGER COUNT, DATATYI t a ready mode nonbl CV(buf, count, datatyp buf	<pre>T(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror TATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR ocking send. be, source, tag, comm, request)</pre>	
INtagmessage tag or MPI_ANY_TAG (integer)INcommcommunicator (handle)	TYPE TYPE INTE Ortran PI_IRSE <typ INTE Start API_IRE OUT IN</typ 	E(MPI_Comm), INTEN E(MPI_Request), IN EGER, OPTIONAL, IN binding END(BUF, COUNT, DA EQER COUNT, DATATYI t a ready mode nonbl CV(buf, count, datatyp buf count	<pre>T(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror TATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR ocking send. be, source, tag, comm, request)</pre>	
IN comm communicator (handle)	TYPE TYPE INTE Fortran PI_IRSE <typ INTE Start OUT IN IN</typ 	E(MPI_Comm), INTEN E(MPI_Request), IN EGER, OPTIONAL, IN binding END(BUF, COUNT, DA De> BUF(*) EGER COUNT, DATATYI t a ready mode nonbl CV(buf, count, datatyp buf count datatype	<pre>T(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror TATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR ocking send. De, source, tag, comm, request)</pre>	
	TYPE TYPE INTE Fortran PI_IRSE <typ INTE Start API_IRE OUT IN IN IN</typ 	E(MPI_Comm), INTENT E(MPI_Request), INT EGER, OPTIONAL, INT binding END(BUF, COUNT, DAT pe> BUF(*) EGER COUNT, DATATYN t a ready mode nonbl CV(buf, count, datatyn buf count datatype source	<pre>T(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror TATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR ocking send. be, source, tag, comm, request) initial address of receive buffer (choice) number of elements in receive buffer (non-negative integer) datatype of each receive buffer element (handle) rank of source or MPI_ANY_SOURCE (integer)</pre>	
OUTrequestcommunication request (handle)	TYPE TYPE INTE Fortran PI_IRSE <typ INTE Start API_IRE OUT IN IN IN IN</typ 	E(MPI_Comm), INTEN E(MPI_Request), INT EGER, OPTIONAL, INT binding END(BUF, COUNT, DAT De> BUF(*) EGER COUNT, DATATYP t a ready mode nonblact CV(buf, count, datatyp buf count datatype source tag	<pre>T(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror TATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR ocking send. De, source, tag, comm, request)</pre>	
	TYPE TYPE INTE Fortran PI_IRSE <typ INTE Start API_IREC OUT IN IN IN IN</typ 	E(MPI_Comm), INTEN E(MPI_Request), INT EGER, OPTIONAL, INT binding END(BUF, COUNT, DAT De> BUF(*) EGER COUNT, DATATYP t a ready mode nonblact CV(buf, count, datatyp buf count datatype source tag	<pre>T(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror TATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR ocking send. De, source, tag, comm, request)</pre>	

Unofficial Draft for Comment Only

47

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
3
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
4
          INTEGER, INTENT(IN) :: count, source, tag
5
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
          TYPE(MPI_Comm), INTENT(IN) :: comm
7
          TYPE(MPI_Request), INTENT(OUT) :: request
8
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     Fortran binding
10
     MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
11
          <type> BUF(*)
12
          INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
13
14
          Start a nonblocking receive.
15
16
     MPI_ISENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype,
17
                     source, recvtag, comm, request)
18
19
       IN
                 sendbuf
                                              initial address of send buffer (choice)
20
       IN
                 sendcount
                                              number of elements in send buffer (non-negative
21
                                              integer)
22
       IN
                 sendtype
                                              datatype of each send buffer element (handle)
23
24
       IN
                 dest
                                              rank of destination (integer)
25
                                              send tag (integer)
       IN
                 sendtag
26
       OUT
                 recvbuf
                                              initial address of receive buffer (choice)
27
28
       IN
                                              number of elements in receive buffer (non-negative
                 recvcount
29
                                              integer)
30
       IN
                                              datatype of each receive buffer element (handle)
                 recvtype
^{31}
       IN
                                              rank of source or MPI_ANY_SOURCE (integer)
                 source
32
33
       IN
                 recvtag
                                              receive tag or MPI_ANY_TAG (integer)
34
       IN
                 comm
                                              communicator (handle)
35
       OUT
                 request
                                              communication request (handle)
36
37
38
     C binding
     int MPI_Isendrecv(const void *sendbuf, int sendcount,
39
40
                     MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf,
41
                     int recvcount, MPI_Datatype recvtype, int source, int recvtag,
42
                     MPI_Comm comm, MPI_Request *request)
43
     Fortran 2008 binding
44
     MPI_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
45
                     recvcount, recvtype, source, recvtag, comm, request, ierror)
46
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
47
48
```

INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source, 1 2 recvtag TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_ISENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, 10 RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, REQUEST, IERROR) 11 <type> SENDBUF(*), RECVBUF(*) 12INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, 13 SOURCE, RECVTAG, COMM, REQUEST, IERROR 1415Initiate a nonblocking communication request for a *send and receive* operation. 1617 MPI_ISENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, 18 request) 19 INOUT 20buf initial address of send and receive buffer (choice) 21IN number of elements in send and receive buffer count 22 (non-negative integer) 23type of elements in send and receive buffer (handle) IN datatype 2425rank of destination (integer) IN dest 26send message tag (integer) IN sendtag 27IN source rank of source or MPI_ANY_SOURCE (integer) 2829 receive message tag or MPI_ANY_TAG (integer) IN recvtag 30 IN communicator (handle) comm 31OUT communication request (handle) request 32 33 34 C binding int MPI_Isendrecv_replace(void *buf, int count, MPI_Datatype datatype, 35 int dest, int sendtag, int source, int recvtag, MPI_Comm comm, 36 MPI_Request *request) 37 38 Fortran 2008 binding 39 MPI_Isendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag, 40 comm, request, ierror) 41 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 42INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag 43 TYPE(MPI_Datatype), INTENT(IN) :: datatype 44 TYPE(MPI_Comm), INTENT(IN) :: comm 45TYPE(MPI_Request), INTENT(OUT) :: request 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4748

Fortran binding

Unofficial Draft for Comment Only

1	MPI_ISENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
2	COMM, REQUEST, IERROR)
3	<type> BUF(*)</type>
4	INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, REQUEST,
5	IERROR
6	

Initiate a nonblocking communication request for a *send and receive* operation. The same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received.

These calls allocate a communication request object and associate it with the request handle (the argument request). The request can be used later to query the status of the communication or wait for its completion.

A nonblocking send call indicates that the system may start copying data out of the send buffer. The sender should not modify any part of the send buffer after a nonblocking send operation is called, until the send completes.

A nonblocking receive call indicates that the system may start writing data into the receive buffer. The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 19.1.10–19.1.20. (End of advice to users.)

22 23 24

7

8

9

10

11

12

16

17

18 19

20

21

3.7.3 Communication Completion

The functions MPI_WAIT and MPI_TEST are used to complete a nonblocking communication. The completion of a send operation indicates that the sender is now free to update the locations in the send buffer (the send operation itself leaves the content of the send buffer unchanged). It does not indicate that the message has been received, rather, it may have been buffered by the communication subsystem. However, if a **synchronous** mode send was used, the completion of the send operation indicates that a matching receive was initiated, and that the message will eventually be received by this matching receive.

The completion of a receive operation indicates that the receive buffer contains the received message, the receiver is now free to access it, and that the status object is set. It does not indicate that the matching send operation has completed (but indicates, of course, that the send was initiated).

We shall use the following terminology: A **null handle** is a handle with value

³⁷ MPI_REQUEST_NULL. A persistent request and the handle to it are **inactive** if the request
 ³⁸ is not associated with any ongoing communication (see Section 3.9). A handle is **active** ³⁹ if it is neither null nor inactive. An **empty** status is a status which is set to return tag
 ⁴⁰ = MPI_ANY_TAG, source = MPI_ANY_SOURCE, error = MPI_SUCCESS, and is also internally
 ⁴¹ configured so that calls to MPI_GET_COUNT, MPI_GET_ELEMENTS, and

MPI_GET_ELEMENTS_X return count = 0 and MPI_TEST_CANCELLED returns false. We
 set a status variable to empty when the value returned by it is not significant. Status is set
 in this way so as to prevent errors due to accesses of stale information.

The fields in a status object returned by a call to MPI_WAIT, MPI_TEST, or any of the other derived functions (MPI_{TEST|WAIT}{ALL|SOME|ANY}), where the request corresponds to a send call, are undefined, with two exceptions: The error status field will contain valid information if the wait or test call returned with MPI_ERR_IN_STATUS; and the returned status can be queried by the call MPI_TEST_CANCELLED.

Error codes belonging to the error class MPI_ERR_IN_STATUS should be returned only by the MPI completion functions that take arrays of MPI_Status. For the functions that take a single MPI_Status argument, the error code is returned by the function, and the value of the MPI_ERROR field in the MPI_Status argument is undefined (see 3.2.5).

INOUT	request	request (handle)		
OUT	status	status object (Status)		

C binding

int MPI_Wait(MPI_Request *request, MPI_Status *status)

Fortran 2008 binding

MPI WAIT(request status)

```
MPI_Wait(request, status, ierror)
   TYPE(MPI_Request), INTENT(INOUT) :: request
   TYPE(MPI_Status) :: status
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
Fortran binding
```

```
MPI_WAIT(REQUEST, STATUS, IERROR)
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

A call to MPI_WAIT returns when the operation identified by request is complete. If the request is an active persistent request, it is marked inactive. Any other type of request is deallocated and the request handle is set to MPI_REQUEST_NULL. MPI_WAIT is a non-local operation.

The call returns, in status, information on the completed operation. The content of the status object for a receive operation can be accessed as described in Section 3.2.5. The status object for a send operation may be queried by a call to MPI_TEST_CANCELLED (see Section 3.8).

One is allowed to call MPI_WAIT with a null or inactive request argument. In this case the operation returns immediately with empty status.

Advice to users. Successful return of MPI_WAIT after a MPI_IBSEND implies that the user send buffer can be reused — i.e., data has been sent out or copied into a buffer attached with MPI_BUFFER_ATTACH. Note that, at this point, we can no longer cancel the send (see Section 3.8). If a matching receive is never posted, then the buffer cannot be freed. This runs somewhat counter to the stated goal of MPI_CANCEL (always being able to free program space that was committed to the communication subsystem). (End of advice to users.)

Advice to implementors. In a multithreaded environment, a call to MPI_WAIT should block only the calling thread, allowing the thread scheduler to schedule another thread for execution. (*End of advice to implementors.*)

 $\mathbf{2}$

```
1
      MPI_TEST(request, flag, status)
2
       INOUT
                  request
                                               communication request (handle)
3
       OUT
                 flag
                                               true if operation completed (logical)
4
5
       OUT
                                               status object (Status)
                 status
6
7
      C binding
8
      int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
9
      Fortran 2008 binding
10
     MPI_Test(request, flag, status, ierror)
11
          TYPE(MPI_Request), INTENT(INOUT) :: request
12
          LOGICAL, INTENT(OUT) :: flag
13
          TYPE(MPI_Status) :: status
14
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
      Fortran binding
17
     MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
18
          INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
19
          LOGICAL FLAG
20
          A call to MPI_TEST returns flag = true if the operation identified by request is complete.
21
     In such a case, the status object is set to contain information on the completed operation.
22
     If the request is an active persistent request, it is marked as inactive. Any other type of
23
     request is deallocated and the request handle is set to MPI_REQUEST_NULL. The call returns
^{24}
      flag = false if the operation identified by request is not complete. In this case, the value of
25
      the status object is undefined. MPI_TEST is a local operation.
26
          The return status object for a receive operation carries information that can be accessed
27
      as described in Section 3.2.5. The status object for a send operation carries information
28
      that can be accessed by a call to MPI_TEST_CANCELLED (see Section 3.8).
29
          One is allowed to call MPI_TEST with a null or inactive request argument. In such a
30
      case the operation returns with flag = true and empty status.
^{31}
          The functions MPI_WAIT and MPI_TEST can be used to complete both sends and
32
      receives.
33
34
           Advice to users.
                               The use of the nonblocking MPI_TEST call allows the user to
35
           schedule alternative activities within a single thread of execution. An event-driven
36
           thread scheduler can be emulated with periodic calls to MPI_TEST. (End of advice to
37
           users.)
38
39
40
     Example 3.11 Simple usage of nonblocking operations and MPI_WAIT.
41
42
43
44
45
46
47
48
```

CALL MPI_COMM_RANK(comm, rank, ierr)	1
IF (rank .EQ. 0) THEN	2
CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)	3 4
**** do some computation to mask latency **** CALL MPI_WAIT(request, status, ierr)	5
ELSE IF (rank .EQ. 1) THEN	6
CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)	7
**** do some computation to mask latency ****	8
CALL MPI_WAIT(request, status, ierr)	9
END IF	10
	11
A request object can be freed using the following MPI procedure.	12
	13
MPI_REQUEST_FREE(request)	14
INOUT request communication request (handle)	15 16
	17
C binding	18
int MPI_Request_free(MPI_Request *request)	19
Fortran 2008 binding	20
MPI_Request_free(request, ierror)	21
TYPE(MPI_Request), INTENT(INOUT) :: request	22
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	23
Forture binding	24
Fortran binding MPI_REQUEST_FREE(REQUEST, IERROR)	$\frac{25}{26}$
INTEGER REQUEST, IERROR	20
	28
MPI_REQUEST_FREE is a local operation. Upon successful return,	29
MPI_REQUEST_FREE sets request to MPI_REQUEST_NULL. For an inactive	30
request representing any type of MPI operation, MPI_REQUEST_FREE shall do the freeing	31
stage of the associated operation during its execution.	32
For a request representing a nonblocking point-to-point or a persistent point-to-point	33
operation, it is permitted (although strongly discouraged) to call MPI_REQUEST_FREE	34
when the request is active. In this special case, MPI_REQUEST_FREE will only mark the	35
request for freeing and MPI will actually do the freeing stage of the associated operation	36
later.	37
The use of this routine for generalized requests is described in Section 13.2.	38
Calling MPI_REQUEST_FREE with an active request representing any other type of	39
MPI operation (e.g., any partitioned operation (see Chapter 4), any collective operation	40
(see Chapter 6), any I/O operation (see Chapter 14), or any request-based RMA operation	41
(see Chapter 12)) is erroneous.	42
Rationale. For point-to-point operations, the MPI_REQUEST_FREE mechanism is	43
provided for reasons of performance and convenience on the sending side. (End of	44
rationale.)	45
, wood to do on j	46
Advice to users Once a request is freed by a call to MPL REOLIEST FREE it is not	47

Advice to users. Once a request is freed by a call to $MPI_REQUEST_FREE$, it is not possible to check for the successful completion of the associated communication with 48

Unofficial Draft for Comment Only

```
1
          calls to MPI_WAIT or MPI_TEST. Also, if an error occurs subsequently during the
\mathbf{2}
          communication, an error code cannot be returned to the user — such an error must
3
          be treated as fatal. An active receive request should never be freed as the receiver
4
          will have no way to verify that the receive has completed and the receive buffer can
5
          be reused. (End of advice to users.)
6
7
     Example 3.12 An example using MPI_REQUEST_FREE.
8
9
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
10
     IF (rank .EQ. 0) THEN
11
        DO i=1,n
12
            CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
13
            CALL MPI_REQUEST_FREE(req, ierr)
14
            CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
15
            CALL MPI_WAIT(req, status, ierr)
16
        END DO
17
     ELSE IF (rank .EQ. 1) THEN
18
        CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
19
        CALL MPI_WAIT(req, status, ierr)
20
        DO I=1,n-1
21
            CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
22
            CALL MPI_REQUEST_FREE(req, ierr)
23
            CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
24
            CALL MPI_WAIT(req, status, ierr)
25
        END DO
26
        CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
27
        CALL MPI_WAIT(req, status, ierr)
28
     END IF
29
30
            Semantics of Nonblocking Communications
     3.7.4
^{31}
32
     The semantics of nonblocking communication is defined by suitably extending the definitions
33
     in Section 3.5.
34
35
     Order Nonblocking communication operations are ordered according to the execution order
36
     of the calls that initiate the communication. The non-overtaking requirement of Section 3.5
37
     is extended to nonblocking communication, with this definition of order being used.
38
39
     Example 3.13 Message ordering for nonblocking operations.
40
     CALL MPI_COMM_RANK(comm, rank, ierr)
41
42
     IF (RANK .EQ. 0) THEN
        CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
43
        CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
44
     ELSE IF (rank .EQ. 1) THEN
45
        CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)
46
        CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
47
     END IF
48
```

CALL MPI_WAIT(r1, status, ierr) CALL MPI_WAIT(r2, status, ierr)

The first send of process zero will match the first receive of process one, even if both messages are sent before process one executes either receive.

Progress A call to MPI_WAIT that completes a receive will eventually terminate and return if a matching send has been started, unless the send is satisfied by another receive. In particular, if the matching send is nonblocking, then the receive should complete even if no call is executed by the sender to complete the send. Similarly, a call to MPI_WAIT that completes a send will eventually return if a matching receive has been started, unless the receive is satisfied by another send, and even if no call is executed to complete the receive.

Example 3.14 An illustration of progress semantics.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK .EQ. 0) THEN
CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, status, ierr)
CALL MPI_WAIT(r, status, ierr)
END IF
```

This code should not deadlock in a correct MPI implementation. The first synchronous send of process zero must complete after process one posts the matching (nonblocking) receive even if process one has not yet reached the completing wait call. Thus, process zero will continue and execute the second send, allowing process one to complete execution.

If an MPI_TEST that completes a receive is repeatedly called with the same arguments, and a matching send has been started, then the call will eventually return flag = true, unless the send is satisfied by another receive. If an MPI_TEST that completes a send is repeatedly called with the same arguments, and a matching receive has been started, then the call will eventually return flag = true, unless the receive is satisfied by another send.

3.7.5 Multiple Completions

It is convenient to be able to wait for the completion of any, some, or all the operations in a list, rather than having to wait for a specific message. A call to MPI_WAITANY or MPI_TESTANY can be used to wait for the completion of one out of several operations. A call to MPI_WAITALL or MPI_TESTALL can be used to wait for all pending operations in a list. A call to MPI_WAITSOME or MPI_TESTSOME can be used to complete all enabled operations in a list.

 $\mathbf{2}$

 24

1	MPI_WAIT	ANY(count, array_of_requests	, index, status)
2	IN	count	list length (non-negative integer)
3	INOUT	array_of_requests	array of requests (array of handles)
4			· - · · · · · · · · · · · · · · · · · ·
5 6	OUT	index	index of handle for operation that completed (integer)
7 8	OUT	status	status object (Status)
9 10 11	C binding int MPI_W	-	quest array_of_requests[], int *index,
12			
13 14 15 16 17 18 19	MPI_Waita INTEG TYPE(INTEG TYPE(ER, INTENT(IN) :: count	sts, index, status, ierror) JT) :: array_of_requests(count)) :: ierror
20	Fortran b	vinding	
21 22		6	STS, INDEX, STATUS, IERROR)
22			STS(*), INDEX, STATUS(MPI_STATUS_SIZE),
24		IERROR	
25	Block	s until one of the operations a	ssociated with the active requests in the array has
26		_	is enabled and can terminate, one is arbitrarily
27			hat request in the array and returns in status the
28 29			e array is indexed from zero in C, and from one in
30	· · ·		rsistent request, it is marked inactive. Any other
31			quest handle is set to MPI_REQUEST_NULL. ain null or inactive handles. If the list contains no
32			l entries are null or inactive), then the call returns
33		ly with index = $MPI_UNDEFIN$	
34			with an array containing multiple entries has the
35			AIT with the array entry indicated by the output
36	value of in	dex (unless the output value of	of index is MPI_UNDEFINED). MPI_WAITANY with
37	an array c	ontaining one active entry is e	quivalent to MPI_WAIT.
38			
39 40			
40			
42			
43			
44			
45			
46			
47			
48			

•••	······································			
	IN	count	list length (non-negative integer)	
	INOUT	array_of_requests	array of requests (array of handles)	
	OUT	index	index of operation that completed or MPI_UNDEFINED if none completed (integer)	
	OUT	flag	true if one of the operations is complete (logical)	
	OUT	status	status object (Status)	

MPI_TESTANY(count, array_of_requests, index, flag, status)

C binding

Fortran 2008 binding

MPI_Testany(count, array_of_requests, index, flag, status, ierror)
 INTEGER, INTENT(IN) :: count
 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
 INTEGER, INTENT(OUT) :: index
 LOGICAL, INTENT(OUT) :: flag
 TYPE(MPI_Status) :: status
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
IERROR

LOGICAL FLAG

Tests for completion of either one or none of the operations associated with active handles. In the former case, it returns flag = true, returns in index the index of this request in the array, and returns in status the status of that operation. If the request is an active persistent request, it is marked as inactive. Any other type of request is deallocated and the handle is set to MPI_REQUEST_NULL. (The array is indexed from zero in C, and from one in Fortran.) In the latter case (no operation completed), it returns flag = false, returns a value of MPI_UNDEFINED in index and status is undefined.

The array may contain null or inactive handles. If the array contains no active handles then the call returns immediately with flag = true, $index = MPI_UNDEFINED$, and an empty status.

If the array of requests contains active handles then the execution of MPI_TESTANY has the same effect as the execution of MPI_TEST with each of the array elements in some arbitrary order, until one call returns flag = true, or all fail. In the former case, index is set to indicate which array element returned flag = true and in the latter case, it is set to MPI_UNDEFINED. MPI_TESTANY with an array containing one active entry is equivalent to MPI_TEST.

 24

1 MPI_WAITALL(count, array_of_requests, array_of_statuses) 2 IN count lists length (non-negative integer) 3 INOUT array_of_requests array of requests (array of handles) 4 5OUT array_of_statuses array of status objects (array of Status) 6 7 C binding 8 int MPI_Waitall(int count, MPI_Request array_of_requests[], 9 MPI_Status array_of_statuses[]) 10 Fortran 2008 binding 11 MPI_Waitall(count, array_of_requests, array_of_statuses, ierror) 12INTEGER, INTENT(IN) :: count 13 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count) 14TYPE(MPI_Status) :: array_of_statuses(*) 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617 Fortran binding 18 MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR) 19INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE, 20*), IERROR 21Blocks until all communication operations associated with active handles in the list 22 complete, and return the status of all these operations (this includes the case where no 23handle in the list is active). Both arrays have the same number of valid entries. The 24 i-th entry in array_of_statuses is set to the return status of the i-th operation. Active 25persistent requests are marked inactive. Requests of any other type are deallocated and the 26corresponding handles in the array are set to MPI_REQUEST_NULL. The list may contain 27null or inactive handles. The call sets to empty the status of each such entry. 28The error-free execution of MPI_WAITALL has the same effect as the execution of 29 MPI_WAIT for each of the array elements in some arbitrary order. MPI_WAITALL with an 30 array of length one is equivalent to MPI_WAIT. 31 When one or more of the communications completed by a call to MPI_WAITALL fail, 32 it is desirable to return specific information on each communication. The function 33 34MPI_WAITALL will return in such case the error code MPI_ERR_IN_STATUS and will set the error field of each status to a specific error code. This code will be MPI_SUCCESS, if the 35 specific communication completed; it will be another specific error code, if it failed; or it can 36 be MPI_ERR_PENDING if it has neither failed nor completed. The function MPI_WAITALL 37 will return MPI_SUCCESS if no request had an error, or will return another error code if it 38 failed for other reasons (such as invalid arguments). In such cases, it will not update the 39 error fields of the statuses. 4041 Rationale. This design streamlines error handling in the application. The application 42code need only test the (single) function result to determine if an error has occurred. It 43 needs to check each individual status only when an error occurred. (End of rationale.) 44 4546 47 48

MPI_TES	TALL(count, array_of_requests	, flag, array_of_statuses)	1	
IN	count	lists length (non-negative integer)	2 3	
INOUT	array_of_requests	array of requests (array of handles)	4	
OUT	flag	(logical)	5	
OUT	array_of_statuses	array of status objects (array of Status)	6	
001	anay_or_statuses	array of status objects (array of status)	7	
C bindin	g		8 9	
	-	<pre>quest array_of_requests[], int *flag,</pre>	10	
	MPI_Status array_of_	statuses[])	11	
Fortran 2	2008 binding		12	
		<pre>sts, flag, array_of_statuses, ierror)</pre>	13	
	GER, INTENT(IN) :: count		14 15	
	(MPI_Request), INTENT(INU CAL, INTENT(OUT) :: flag	UT) :: array_of_requests(count)	16	
	(MPI_Status) :: array_of_	statuses(*)	17	
	GER, OPTIONAL, INTENT(OUT		18	
Fortran l	ainding		19	
	8	STS, FLAG, ARRAY_OF_STATUSES, IERROR)	20 21	
		<pre>STS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,</pre>	21	
	*), IERROR		23	
LOGI	CAL FLAG		24	
Retur	Returns $flag = true$ if all communications associated with active handles in the array			
-		here no handle in the list is active). In this case, each	26 27	
	-	ve request is set to the status of the corresponding	28	
-		re marked inactive. Requests of any other type are adles in the array are set to MPI_REQUEST_NULL.	29	
		null or inactive handle is set to empty.	30	
Other	wise, $flag = false$ is returned,	no request is modified and the values of the status	31	
	undefined. This is a local op		32 33	
		ecution of MPI_TESTALL are handled in the same	34	
manner as	errors in MPI_WAITALL.		35	
			36	
			37	
			38	
			39 40	
			41	
			42	
			43	
			44	
			45	
			46 47	
			48	

1 2	MPI_WAIT	SOME(incount, array_of_reque	ests, outcount, array_of_indices, array_of_statuses)	
3	IN	incount	length of array_of_requests (non-negative integer)	
4 5	INOUT	array_of_requests	array of requests (array of handles)	
6	OUT	outcount	number of completed requests (integer)	
7 8 9	OUT	array_of_indices	array of indices of operations that completed (array of integers)	
9 10 11	Ουτ	array_of_statuses	array of status objects for operations that completed (array of Status)	
12	~			
13 14 15 16	C binding int MPI_W	-		
17 18 19 20 21 22 23 24 25	<pre>Fortran 2008 binding MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,</pre>			
26 27 28 29 30	<pre>Fortran binding MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,</pre>			
 31 32 33 34 35 36 37 38 39 40 	completed. have compl indices of t from zero i array array requests ar and the ass	Returns in outcount the num leted. Returns in the first out chese operations (index within in C and from one in Fortran _of_status the status for these e marked as inactive. Any ot sociated handle is set to MPI_ list contains no active handles	tions associated with active handles in the list have ber of requests from the list array_of_requests that atcount locations of the array array_of_indices the in the array array_of_requests; the array is indexed in). Returns in the first outcount locations of the completed operations. Completed active persistent her type or request that completed is deallocated, REQUEST_NULL. s, then the call returns immediately with outcount	
41 42 43 44 45 46 47 48	When it is desiral outcount, a all commu MPI_ERR_II success or	one or more of the communi- ble to return specific informa- rray_of_indices and array_of_s nications that have succeede N_STATUS and the error field to indicate the specific error	cations completed by MPI_WAITSOME fails, then tion on each communication. The arguments statuses will be adjusted to indicate completion of d or failed. The call will return the error code d of each status returned will be set to indicate that occurred. The call will return MPI_SUCCESS vill return another error code if it failed for other	

reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

MPI_TESTSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

			6	
IN	incount	length of array_of_requests (non-negative integer)	7	
INOUT	array_of_requests	array of requests (array of handles)	8	
OUT	outcount	number of completed requests (integer)	9 10	
OUT	array_of_indices	array of indices of operations that completed (array	11	
001		of integers)	12	
		с ,	13	
OUT	array_of_statuses	array of status objects for operations that completed (array of Status)	14	
		(array of Status)	15	
Chindin	-		16	
C binding		_Request array_of_requests[],	17	
IIIC MFI_I	int *outcount, int a		18	
	MPI_Status array_of_	·	19	
_	·		20	
	008 binding		21 22	
MP1_Tests		quests, outcount, array_of_indices,	22	
array_of_statuses, ierror)				
<pre>INTEGER, INTENT(IN) :: incount TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)</pre>				
<pre>TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount) INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)</pre>				
			27	
	ER, OPTIONAL, INTENT(OUT)		28	
			29	
Fortran b	<u> </u>		30	
MPI_IESIS	ARRAY_OF_STATUSES, I	QUESTS, OUTCOUNT, ARRAY_OF_INDICES,	31	
INTEG		JESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),	32	
INTEG		PI_STATUS_SIZE, *), IERROR	33 34	
			35	
		pt that it returns immediately. If no operation has	36	
-		ere is no active handle in the list it returns outcount	37	
= MPI_UNI		on, which returns immediately, whereas	38	
		munication completes, if it was passed a list that	39	
		the calls fulfill a fairness requirement: If a request	40	
		t of requests passed to MPI_WAITSOME or	41	
		has been posted, then the receive will eventually	42	

MPI_TESTSOME, and a matching send has been posted, then the receive will eventually succeed, unless the send is satisfied by another receive; and similarly for send requests.

Errors that occur during the execution of $\mathsf{MPI_TESTSOME}$ are handled as for $\mathsf{MPI_WAITSOME}.$

Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use of MPI_TESTANY. The former returns information on all completed communications,

Unofficial Draft for Comment Only

1 2

 $\frac{3}{4}$

5

43

44

45

46

```
1
          with the latter, a new call is required for each communication that completes.
\mathbf{2}
          A server with multiple clients can use MPI_WAITSOME so as not to starve any client.
3
          Clients send messages to the server with service requests. The server calls
4
          MPI_WAITSOME with one receive request for each client, and then handles all receives
5
          that completed. If a call to MPI_WAITANY is used instead, then one client could starve
6
          while requests from another client always sneak in first. (End of advice to users.)
7
8
                                    MPI_TESTSOME should complete as many pending com-
          Advice to implementors.
9
          munications as possible. (End of advice to implementors.)
10
11
     Example 3.15 Client-server code (starvation can occur).
12
13
     CALL MPI_COMM_SIZE(comm, size, ierr)
14
     CALL MPI_COMM_RANK(comm, rank, ierr)
15
     IF (rank .GT. 0) THEN
                                       ! client code
16
        DO WHILE(.TRUE.)
17
            CALL MPI_ISEND(a, n, MPI_REAL, 0, tag,
                                                        comm, request, ierr)
18
            CALL MPI_WAIT(request, status, ierr)
19
        END DO
20
     ELSE
                    ! rank=0 -- server code
21
        DO i=1, size-1
22
            CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag, &
23
                             comm, request_list(i), ierr)
24
        END DO
25
        DO WHILE(.TRUE.)
26
            CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
27
            CALL DO_SERVICE(a(1, index)) ! handle one message
28
            CALL MPI_IRECV(a(1, index), n, MPI_REAL, index, tag, &
29
                             comm, request_list(index), ierr)
30
        END DO
31
     END IF
32
33
34
     Example 3.16 Same code, using MPI_WAITSOME.
35
     CALL MPI_COMM_SIZE(comm, size, ierr)
36
     CALL MPI_COMM_RANK(comm, rank, ierr)
37
     IF (rank .GT. 0) THEN
                                       ! client code
38
        DO WHILE(.TRUE.)
39
            CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
40
            CALL MPI_WAIT(request, status, ierr)
41
        END DO
42
     ELSE
                    ! rank=0 -- server code
43
        DO i=1, size-1
44
            CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag, &
45
                             comm, request_list(i), ierr)
46
        END DO
47
        DO WHILE(.TRUE.)
48
```

```
CALL MPI_WAITSOME(size, request_list, numdone, &
indices, statuses, ierr)
DO i=1,numdone
CALL DO_SERVICE(a(1, indices(i)))
CALL MPI_IRECV(a(1, indices(i)), n, MPI_REAL, 0, tag, &
comm, request_list(indices(i)), ierr)
END DO
END DO
END DO
END IF
```

3.7.6 Non-Destructive Test of status

This call is useful for accessing the information associated with a request, without freeing the request (in case the user is expected to access it later). It allows one to layer libraries more conveniently, since multiple layers of software may access the same completed request and extract from it the status information.

MPI_REQUEST_GET_STATUS(request, flag, status)				
IN IN	request	request (handle)	19	
	request	,	20	
OUT	flag	boolean flag, same as from MPI_TEST (logical)	21	
OUT	status	status object if flag is true (Status)	22	
			23	
C binding	r		24	
-	-	quest request, int *flag,	25	
IIIC III I_IC	MPI_Status *status)	quest request, int *riag,	26	
	MFI_Status *Status)		27	
Fortran 2	008 binding		28	
MPI_Request_get_status(request, flag, status, ierror)				
TYPE(MPI_Request), INTENT(IN) :: request				
	AL, INTENT(OUT) :: flag		31	
TYPE(MPI_Status) :: status				
INTEG	ER, OPTIONAL, INTENT(OUT) :: ierror	33	
			34	
Fortran b	U		35	
	ST_GET_STATUS(REQUEST, F		36	
	ER REQUEST, STATUS(MPI_S	TATUS_SIZE), IERROR	37	
LOGIC	AL FLAG		38	
Sets fl	ag = true if the operation is	s complete, and, if so, returns in status the request	39	
	-	t does not deallocate or inactivate the request; a	40	
	, , , , , , , , , , , , , , , , , , , ,	build be executed with that request. It sets flag =	41	
-	e operation is not complete.	sure se excerted with that request. It sets hag -	42	
	o operation is not complete.		4.9	

One is allowed to call MPI_REQUEST_GET_STATUS with a null or inactive request argument. In such a case the operation returns with flag = true and empty status.

```
45
46
47
```

43

44

1

 $\mathbf{2}$

3

4

5

6

7 8

9

10 11

12

13

14

15

1	3.8 Pro	be and Cancel		
2 3 4 5 6 7 8 9 10 11 12 13	The MPI_PROBE, MPI_IPROBE, MPI_MPROBE, and MPI_IMPROBE operations allow in- coming messages to be checked for, without actually receiving them. The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by status). In particular, the user may allocate memory for the receive buffer, according to the length of the probed message. The MPI_CANCEL operation allows pending communications to be cancelled. This is required for cleanup. Posting a send or a receive ties up user resources (send or receive buffers), and a cancel may be needed to free these resources gracefully. Cancelling a send request by calling MPI_CANCEL is deprecated. Cancelling a sendrecv request by calling MPI_CANCEL is not allowed.			
14 15	3.8.1 Pro	obe		
15 16				
17 18	MPI_IPRO	BE(source, tag, comm, flag, st	catus)	
19	IN	source	rank of source or MPI_ANY_SOURCE (integer)	
20	IN	tag	message tag or $MPI_ANY_TAG\xspace$ (integer)	
21 22	IN	comm	communicator (handle)	
23	OUT	flag	(logical)	
24	OUT	status	status object (Status)	
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39	Fortran 2 MPI_Iprob INTEG TYPE(LOGIC TYPE(INTEG Fortran b MPI_IPROB	<pre>Probe(int source, int ta MPI_Status *status) 2008 binding be(source, tag, comm, fla GER, INTENT(IN) :: source (MPI_Comm), INTENT(IN) :: CAL, INTENT(OUT) :: flag (MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) binding BE(SOURCE, TAG, COMM, FLA</pre>	, tag comm) :: ierror G, STATUS, IERROR)	
40		GER SOURCE, TAG, COMM, ST CAL FLAG	ATUS(MPI_STATUS_SIZE), IERROR	
41 42 43 44 45 46 47 48	MPI_I that can b and comm MPI_RECV returns in a	PROBE(source, tag, comm, fl e received and that matches . The call matches the same : /(, source, tag, comm, statu	ag, status) returns flag = true if there is a message the pattern specified by the arguments source, tag, message that would have been received by a call to is) executed at the same point in the program, and ild have been returned by MPI_RECV(). Otherwise, status undefined.	

If MPI_IPROBE returns flag = true, then the content of the status object can be subsequently accessed as described in Section 3.2.5 to find the source, tag and length of the probed message.

MPI_IPROBE is a local procedure since its return does not depend on MPI calls in other MPI processes, which is marked with the prefix I (for immediate).

A subsequent receive executed with the same communicator, and the source and tag returned in status by MPI_IPROBE will receive the message that was matched by the probe, if no other intervening receive occurs after the probe, and the send is not successfully cancelled before the receive. If the receiving process is multithreaded, it is the user's responsibility to ensure that the last condition holds.

The source argument of MPI_PROBE can be MPI_ANY_SOURCE, and the tag argument can be MPI_ANY_TAG, so that one can probe for messages from an arbitrary source and/or with an arbitrary tag. However, a specific communication context must be provided with the comm argument.

It is not necessary to receive a message immediately after it has been probed for, and the same message may be probed for several times before it is received.

A probe with MPI_PROC_NULL as source returns flag = true, and the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0; see Section 3.10.

MPI_PROBE(source, tag, comm, status)

IN	source	rank of source or MPI_ANY_SOURCE (integer)
IN	tag	message tag or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

C binding

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Fortran 2008 binding

```
MPI_Probe(source, tag, comm, status, ierror)
    INTEGER, INTENT(IN) :: source, tag
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
```

INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_PROBE behaves like MPI_IPROBE except that it is a non-local call that returns only after a matching message has been found.

The MPI implementation of MPI_PROBE and MPI_IPROBE needs to guarantee progress: 43 if a call to MPI_PROBE has been issued by a process, and a send that matches the probe 44 has been initiated by some process, then the call to MPI_PROBE will return, unless the 45 message is received by another concurrent receive operation (that is executed by another 46 thread at the probing process). Similarly, if a process busy waits with MPI_IPROBE and a 47 matching message has been issued, then the call to MPI_IPROBE will eventually return flag 48

Unofficial Draft for Comment Only

1

2

3

4

5 6

 $\overline{7}$

8 9

10

11

12

13 14

15

16

17

18

19 20

29

30

31

32

33

34

35

36 37

38

39

40

41

```
1
     = true unless the message is received by another concurrent receive operation or matched
\mathbf{2}
     by a concurrent matched probe.
3
     Example 3.17 Use probe to wait for an incoming message.
4
5
          CALL MPI_COMM_RANK(comm, rank, ierr)
6
          IF (rank .EQ. 0) THEN
7
             CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
8
          ELSE IF (rank .EQ. 1) THEN
9
             CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
10
          ELSE IF (rank .EQ. 2) THEN
11
             DO i=1,2
12
                CALL MPI_PROBE(MPI_ANY_SOURCE, 0, &
13
                                 comm, status, ierr)
14
                IF (status(MPI_SOURCE) .EQ. 0) THEN
15
                    CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)
     100
16
                ELSE
17
     200
                    CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr)
18
                END IF
19
             END DO
20
          END IF
21
22
     Each message is received with the right type.
23
^{24}
     Example 3.18 A similar program to the previous example, but now it has a problem.
25
26
          CALL MPI_COMM_RANK(comm, rank, ierr)
27
          IF (rank .EQ. 0) THEN
28
             CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
29
          ELSE IF (rank .EQ. 1) THEN
30
             CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
31
          ELSE IF (rank .EQ. 2) THEN
32
             DO i=1,2
33
                CALL MPI_PROBE(MPI_ANY_SOURCE, 0, &
34
                                 comm, status, ierr)
35
                IF (status(MPI_SOURCE) .EQ. 0) THEN
36
     100
                    CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE, &
37
                                    0, comm, status, ierr)
38
                ELSE
39
     200
                    CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE, &
40
                                    0, comm, status, ierr)
41
                END IF
42
             END DO
43
          END IF
44
45
         In Example 3.18, the two receive calls in statements labeled 100 and 200 in Example 3.17
46
     are slightly modified, using MPI_ANY_SOURCE as the source argument. The program is now
47
     incorrect: the receive operation may receive a message that is distinct from the message
```

probed by the preceding call to MPI_PROBE.

Advice to users. In a multithreaded MPI program, MPI_PROBE and MPI_IPROBE might need special care. If a thread probes for a message and then immediately posts a matching receive, the receive may match a message other than that found by the probe since another thread could concurrently receive that original message [33]. MPI_MPROBE and MPI_IMPROBE solve this problem by matching the incoming message so that it may only be received with MPI_MRECV or MPI_IMRECV on the corresponding message handle. (*End of advice to users.*)

Advice to implementors. A call to MPI_PROBE(source, tag, comm, status) will match the message that would have been received by a call to MPI_RECV(..., source, tag, comm, status) executed at the same point. Suppose that this message has source s, tag t and communicator c. If the tag argument in the probe call has value MPI_ANY_TAG then the message probed will be the earliest pending message from source s with communicator c and any tag; in any case, the message probed will be the earliest pending message from source s with tag t and communicator c (this is the message that would have been received, so as to preserve message order). This message continues as the earliest pending message from source s with tag t and communicator c, until it is received. A receive operation subsequent to the probe that uses the same communicator as the probe and uses the tag and source values returned by the probe, must receive this message, unless it has already been received by another receive operation. (*End of advice to implementors.*)

3.8.2 Matching Probe

The function MPI_PROBE checks for incoming messages without receiving them. Since the list of incoming messages is global among the threads of each MPI process, it can be hard to use this functionality in threaded environments [33, 30].

Like MPI_PROBE and MPI_IPROBE, the MPI_MPROBE and MPI_IMPROBE operations allow incoming messages to be queried without actually receiving them, except that MPI_MPROBE and MPI_IMPROBE provide a mechanism to receive the specific message that was matched regardless of other intervening probe or receive operations. This gives the application an opportunity to decide how to receive the message, based on the information returned by the probe. In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.

MPI_IMPROBE(source, tag, comm, flag, message, status) 36 37 IN rank of source or MPI_ANY_SOURCE (integer) source 38 IN message tag or MPI_ANY_TAG (integer) tag 39 IN comm communicator (handle) 40 41 OUT flag flag (logical) 42OUT returned message (handle) message 43 OUT status status object (Status) 444546C binding 47

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6 7

8

9

10

11

12

13

14

15

16

17

18

19

20

21 22

23 24

25

26

27

28

29

30

 31

32

33

34 35

1 Fortran 2008 binding $\mathbf{2}$ MPI_Improbe(source, tag, comm, flag, message, status, ierror) 3 INTEGER, INTENT(IN) :: source, tag 4 TYPE(MPI_Comm), INTENT(IN) :: comm 5LOGICAL, INTENT(OUT) :: flag 6 TYPE(MPI_Message), INTENT(OUT) :: message 7 TYPE(MPI_Status) :: status 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 Fortran binding 10 MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR) 11 INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR 12LOGICAL FLAG 13 14MPI_IMPROBE(source, tag, comm, flag, message, status) returns flag = true if there is 15a message that can be received and that matches the pattern specified by the arguments 16source, tag, and comm. The call matches the same message that would have been received 17by a call to MPI_RECV(..., source, tag, comm, status) executed at the same point in the 18program and returns in status the same value that would have been returned by MPI_RECV. 19In addition, it returns in message a handle to the matched message. Otherwise, the call 20returns flag = false, and leaves status and message undefined. 21MPI_IMPROBE is a local procedure. According to the definitions in Section 2.4.2 and 22in contrast to MPI_IPROBE, it is a nonblocking procedure because it is the initialization of 23a matched receive operation. 24 A matched receive (MPI_MRECV or MPI_IMRECV) executed with the message han-25dle will receive the message that was matched by the probe. Unlike MPI_IPROBE, no 26other probe or receive operation may match the message returned by MPI_IMPROBE. 27Each message returned by MPI_IMPROBE must be received with either MPI_MRECV or 28MPI_IMRECV. 29The source argument of MPI_IMPROBE can be MPI_ANY_SOURCE, and the tag argu-30 ment can be MPI_ANY_TAG, so that one can probe for messages from an arbitrary source 31 and/or with an arbitrary tag. However, a specific communication context must be provided 32 with the comm argument. 33 A synchronous send operation that is matched with MPI_IMPROBE or MPI_MPROBE 34 will complete successfully only if both a matching receive is posted with MPI_MRECV or 35 MPI_IMRECV, and the receive operation has started to receive the message sent by the 36 synchronous send. 37 There is a special predefined message: MPI_MESSAGE_NO_PROC, which is a message 38which has MPI_PROC_NULL as its source process. The predefined constant 39 MPI_MESSAGE_NULL is the value used for invalid message handles. 40A matching probe with source = MPI_PROC_NULL returns flag = true, message = 41 MPI_MESSAGE_NO_PROC, and the status object returns source = MPI_PROC_NULL, tag = 42MPI_ANY_TAG, and count = 0; see Section 3.10. It is not necessary to call MPI_MRECV or 43MPI_IMRECV with MPI_MESSAGE_NO_PROC, but it is not erroneous to do so. 4445Rationale. MPI_MESSAGE_NO_PROC was chosen instead of MPI_MESSAGE_PROC_NULL to avoid possible confusion as another null handle con-4647 stant. (End of rationale.) 48

0.01 1100			
MPI_MPROBE(source, tag, comm, message, status)			
IN	source	rank of source or MPI_ANY_SOURCE (integer)	
IN	tag	message tag or $MPI_ANY_TAG\xspace$ (integer)	
IN	comm	communicator (handle)	
OUT	message	returned message (handle)	
OUT	status	status object (Status)	
	Mprobe(int source, int tag MPI_Status *status)	g, MPI_Comm comm, MPI_Message *message,	
<pre>Fortran 2008 binding MPI_Mprobe(source, tag, comm, message, status, ierror) INTEGER, INTENT(IN) :: source, tag TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Message), INTENT(OUT) :: message TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>			
Fortran binding MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR) INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR			
MPI_MPROBE behaves like MPI_IMPROBE except that it is a blocking call that return only after a matching message has been found. The implementation of MPI_MPROBE and MPI_IMPROBE needs to guarantee progra- in the same way as in the case of MPI_PROBE and MPI_IPROBE. According to the definitions in Section 2.4.2, MPI_MPROBE is incomplete. It is also non-local procedure			

non-local procedure.

This is one of the exceptions in which incomplete procedures are Advice to users. non-local. (End of advice to users.)

3.8.3 Matched Receives

The functions MPI_MRECV and MPI_IMRECV receive messages that have been previously matched by a matching probe (Section 3.8.2).

 $\mathbf{2}$

1 MPI_MRECV(buf, count, datatype, message, status) 2 OUT buf initial address of receive buffer (choice) 3 IN count number of elements in receive buffer (non-negative 4 integer) 56 IN datatype of each receive buffer element (handle) datatype 7 INOUT message message (handle) 8 OUT status status object (Status) 9 10 11 C binding int MPI_Mrecv(void *buf, int count, MPI_Datatype datatype, 12MPI_Message *message, MPI_Status *status) 13 14Fortran 2008 binding 15MPI_Mrecv(buf, count, datatype, message, status, ierror) 16TYPE(*), DIMENSION(..) :: buf 17 INTEGER, INTENT(IN) :: count 18 TYPE(MPI_Datatype), INTENT(IN) :: datatype 19 TYPE(MPI_Message), INTENT(INOUT) :: message 20TYPE(MPI_Status) :: status 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 Fortran binding 23MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR) 24 25<type> BUF(*) 26INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR 27This call receives a message matched by a matching probe operation (Section 3.8.2). 28The receive buffer consists of the storage containing **count** consecutive elements of the 29 type specified by datatype, starting at address buf. The length of the received message must 30 be less than or equal to the length of the receive buffer. An overflow error occurs if all 31 incoming data does not fit, without truncation, into the receive buffer. 32 If the message is shorter than the receive buffer, then only those locations corresponding 33 to the (shorter) message are modified. 34 On return from this function, the message handle is set to MPI_MESSAGE_NULL. All 35 errors that occur during the execution of this operation are handled according to the error 36 handler set for the communicator used in the matching probe call that produced the message 37 handle. 38 If MPI_MRECV is called with MPI_MESSAGE_NO_PROC as the message argument, the 39 call returns immediately with the status object set to $source = MPI_PROC_NULL$, 40 $tag = MPI_ANY_TAG$, and count = 0, as if a receive from MPI_PROC_NULL was issued (see 41 Section 3.10). A call to MPI_MRECV with MPI_MESSAGE_NULL is erroneous. 4243 44 4546 47 48

ECV(buf, count, datatype, message, request)			
buf	initial address of receive buffer (choice)		
count	number of elements in receive buffer (non-negative integer)		
datatype	datatype of each receive buffer element (handle)		

MPI_IMRI

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (non-negatinteger)
IN	datatype	datatype of each receive buffer element (handle
INOUT	message	message (handle)
OUT	request	communication request (handle)

C binding

int MPI_Imred	v(void *buf	, int count	, MPI_Dataty	pe datatype,
	MPI_Message	*message,	MPI_Request	<pre>*request)</pre>

Fortran 2008 binding

<pre>MPI_Imrecv(buf, count, datatype, message, request, ierror)</pre>	
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	
INTEGER, INTENT(IN) :: count	
TYPE(MPI_Datatype), INTENT(IN) :: datatype	*
TYPE(MPI_Message), INTENT(INOUT) :: message	
TYPE(MPI_Request), INTENT(OUT) :: request	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	

Fortran binding

MPI_	_IMRECV(H	BUF,	COUNT,	DATATY	7PE, MES	SAGE,	REQUE	EST, I	IERROR)	
	<type> H</type>	BUF(*)							
	INTEGER	COUN	T, DATA	ATYPE,	MESSAGE	, REQI	UEST,	IERR	OR	

MPI_IMRECV is the nonblocking variant of MPI_MRECV and starts a nonblocking receive of a matched message. Completion semantics are similar to MPI_IRECV as described in Section 3.7.2. On return from this function, the message handle is set to MPI_MESSAGE_NULL.

If MPI_IMRECV is called with MPI_MESSAGE_NO_PROC as the message argument, the call returns immediately with a request object which, when completed, will yield a status object set to source = MPI_PROC_NULL, $tag = MPI_ANY_TAG$, and count = 0, as if a receive from MPI_PROC_NULL was issued (see Section 3.10). A call to MPI_IMRECV with MPI_MESSAGE_NULL is erroneous.

Advice to implementors. If reception of a matched message is started with MPI_IMRECV, then it is possible to cancel the returned request with MPI_CANCEL. If MPI_CANCEL succeeds, the matched message must be found by a subsequent message probe (MPI_PROBE, MPI_IPROBE, MPI_MPROBE, or MPI_IMPROBE), received by a subsequent receive operation or cancelled by the sender. See Section 3.8.4 for details about MPI_CANCEL. The cancellation of operations initiated with MPI_IMRECV may fail. (End of advice to implementors.)

```
1
      3.8.4
            Cancel
2
3
4
      MPI_CANCEL(request)
5
       IN
                  request
                                              communication request (handle)
6
7
     C binding
8
      int MPI_Cancel(MPI_Request *request)
9
10
     Fortran 2008 binding
11
     MPI_Cancel(request, ierror)
12
          TYPE(MPI_Request), INTENT(IN) :: request
13
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     Fortran binding
15
     MPI_CANCEL(REQUEST, IERROR)
16
          INTEGER REQUEST, IERROR
17
18
          A call to MPI_CANCEL marks for cancellation a pending, nonblocking communica-
19
      tion operation (send or receive). Cancelling a send request by calling MPI_CANCEL is
20
      deprecated. The cancel call is local. It returns immediately, possibly before the communi-
21
      cation is actually cancelled. It is still necessary to call MPI_REQUEST_FREE, MPI_WAIT or
22
      MPI_TEST (or any of the derived operations) with the cancelled request as argument after
23
      the call to MPI_CANCEL. If a communication is marked for cancellation, then a MPI_WAIT
24
      call for that communication is guaranteed to return, irrespective of the activities of other
25
      processes (i.e., MPI_WAIT behaves as a local function); similarly if MPI_TEST is repeatedly
26
      called in a busy wait loop for a cancelled communication, then MPI_TEST will eventually
27
      be successful.
28
          MPI_CANCEL can be used to cancel a communication that uses a persistent request (see
29
      Section 3.9), in the same way it is used for nonpersistent requests. Cancelling a persistent
30
      send request by calling MPL CANCEL is deprecated. A successful cancellation cancels the
^{31}
      active communication, but not the request itself. After the call to MPI_CANCEL and the
32
      subsequent call to MPI_WAIT or MPI_TEST, the request becomes inactive and can be
33
      activated for a new communication.
34
        The successful cancellation of a buffered send frees the buffer space occupied by the
35
      pending message. Cancelling a buffered send request by calling MPI_CANCEL is deprecated.
36
          Either the cancellation succeeds, or the communication succeeds, but not both. If a
37
```

send is marked for cancellation, which is deprecated, then it must be the case that either the send completes normally, in which case the message sent was received at the destination process, or that the send is successfully cancelled, in which case no part of the message was received at the destination. Then, any matching receive has to be satisfied by another send. If a receive is marked for cancellation, then it must be the case that either the receive completes normally, or that the receive is successfully cancelled, in which case no part of the receive buffer is altered. Then, any matching send has to be satisfied by another receive.

⁴⁴ If the operation has been cancelled, then information to that effect will be returned in ⁴⁵ the status argument of the operation that completes the communication.

Rationale. Although the IN request handle parameter should not need to be passed by reference, the C binding has listed the argument type as MPI_Request* since MPI-

Unofficial Draft for Comment Only

38

39

40

41

42

43

47

1.0. This function signature therefore cannot be changed without breaking existing MPI applications. (*End of rationale.*)

MPI_TE	ST_CANCELLED(status, flag	g)
IN	status	status object (Status)
OUT	flag	(logical)
C bindi int MPI		PI_Status *status, int *flag)
	2008 binding	
	t_cancelled(status, fla E(MPI_Status), INTENT(I	-
	ICAL, INTENT(OUT) :: fl	
INT	EGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran	binding	
MPI_TES	T_CANCELLED(STATUS, FLA	G, IERROR)
	EGER STATUS(MPI_STATUS_ ICAL FLAG	SIZE), IERROR
Ret	$\operatorname{urns} flag = true \operatorname{if} \operatorname{the} \operatorname{commu}$	inication associated with the status object was cancelled

successfully. In such a case, all other fields of status (such as count or tag) are undefined. Returns flag = false, otherwise. If a receive operation might be cancelled then one should call MPI_TEST_CANCELLED first, to check whether the operation was cancelled, before checking on the other fields of the return status.

Advice to users. Cancel can be an expensive operation that should be used only exceptionally. (End of advice to users.)

Advice to implementors. If a send operation uses an "eager" protocol (data is transferred to the receiver before a matching receive is posted), then the cancellation of this send may require communication with the intended receiver in order to free allocated buffers. On some systems this may require an interrupt to the intended receiver. Note that, while communication may be needed to implement MPI_CANCEL, this is still a local operation, since its completion does not depend on the code executed by other processes. If processing is required on another process, this should be transparent to the application (hence the need for an interrupt and an interrupt handler). (*End of advice to implementors.*)

3.9 Persistent Communication Requests

Often a communication with the same argument list (with the exception of the buffer contents) is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a **persistent** communication request once and, then, repeatedly

Unofficial Draft for Comment Only

1 using the request to initiate and complete operations. In the case of point-to-point commu- $\mathbf{2}$ nication, the persistent request thus created can be thought of as a communication port or 3 a "half-channel." It does not provide the full functionality of a conventional channel, since 4 there is no binding of the send port to the receive port. This construct allows reduction 5of the overhead for communication between the process and communication controller, but 6 not of the overhead for communication between one communication controller and another. 7 It is not necessary that messages sent with a persistent point-to-point request be received 8 by a receive operation using a persistent point-to-point request, or vice versa.

⁹ There are also collective communication persistent operations defined in Section 6.13 and Section 8.8. The remainder of this section covers the point-to-point persistent initialization operations and the start routines, which are used for both point-to-point and collective persistent communication.

A persistent point-to-point communication request is created using one of the five
 following calls. These point-to-point persistent calls involve no communication.

15 16 17

MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)

18	IN	buf	initial address of send buffer (choice)
19 20	IN	count	number of elements sent (non-negative integer)
21	IN	datatype	type of each element (handle)
22	IN	dest	rank of destination (integer)
23 24	IN	tag	message tag (integer)
24 25	IN	comm	communicator (handle)
26	OUT	request	communication request (handle)

²⁸ C binding

29 30

 31

48

27

```
32 Fortran 2008 binding
```

```
MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
33
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
34
         INTEGER, INTENT(IN) :: count, dest, tag
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         TYPE(MPI_Comm), INTENT(IN) :: comm
37
         TYPE(MPI_Request), INTENT(OUT) :: request
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     Fortran binding
41
     MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
42
         <type> BUF(*)
43
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
44
         Creates a persistent communication request for a standard mode send operation, and
45
     binds to it all the arguments of a send operation.
46
47
```

IN	buf	initial address of send buffer (choice)	
IN	count	number of elements sent (non-negative integer)	
IN	datatype	type of each element (handle)	
IN	dest	rank of destination (integer)	
IN	tag	message tag (integer)	
IN	comm	communicator (handle)	
OUT	request	communication request (handle)	
bindi	0	wild the first south MDT Detators latestand	
nt MPI.		roid *buf, int count, MPI_Datatype datatype, t tag, MPI_Comm comm, MPI_Request *request)	
		s sug, in i_comm comm, in i_noquose sicquose,	
	2008 binding	datatype, dest, tag, comm, request, ierror)	
		, INTENT(IN), ASYNCHRONOUS :: buf	
	EGER, INTENT(IN) ::		
	V1	TENT(IN) :: datatype	
	E(MPI_Comm), INTENT		
	E(MPI_Request), INI EGER, OPTIONAL, INI	ENT(OUT) :: request	
	binding		
		DATATVDE DECT TAC COMM DECUECT LEDDOD)	
		DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	
<ty]< td=""><td>pe> BUF(*)</td><td>DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) E, DEST, TAG, COMM, REQUEST, IERROR</td><td></td></ty]<>	pe> BUF(*)	DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) E, DEST, TAG, COMM, REQUEST, IERROR	
<ty] INT</ty] 	pe> BUF(*) EGER COUNT, DATATYP	E, DEST, TAG, COMM, REQUEST, IERROR	
<ty] INT</ty] 	pe> BUF(*) EGER COUNT, DATATYP		
<tyj INT Crea</tyj 	pe> BUF(*) EGER COUNT, DATATYF ates a persistent comm	E, DEST, TAG, COMM, REQUEST, IERROR unication request for a buffered mode send.	
<tyj INT Crea PI_SSE</tyj 	pe> BUF(*) EGER COUNT, DATATYF ates a persistent comm END_INIT(buf, count, c	PE, DEST, TAG, COMM, REQUEST, IERROR nunication request for a buffered mode send. Hatatype, dest, tag, comm, request)	
<tyj INT Crea PI_SSE IN</tyj 	pe> BUF(*) EGER COUNT, DATATYF ates a persistent comm END_INIT(buf, count, c buf	PE, DEST, TAG, COMM, REQUEST, IERROR nunication request for a buffered mode send. Hatatype, dest, tag, comm, request) initial address of send buffer (choice)	
<tyj INT Crea PI_SSE</tyj 	pe> BUF(*) EGER COUNT, DATATYF ates a persistent comm END_INIT(buf, count, c	PE, DEST, TAG, COMM, REQUEST, IERROR nunication request for a buffered mode send. Hatatype, dest, tag, comm, request)	
<tyj INT Crea PI_SSE IN IN</tyj 	pe> BUF(*) EGER COUNT, DATATYF ates a persistent comm END_INIT(buf, count, c buf	PE, DEST, TAG, COMM, REQUEST, IERROR nunication request for a buffered mode send. Hatatype, dest, tag, comm, request) initial address of send buffer (choice)	
<tyj INT Crea PI_SSE IN IN IN</tyj 	pe> BUF(*) EGER COUNT, DATATYP ates a persistent comm END_INIT(buf, count, c buf count	 DEST, TAG, COMM, REQUEST, IERROR nunication request for a buffered mode send. datatype, dest, tag, comm, request) initial address of send buffer (choice) number of elements sent (non-negative integer) 	
<tyj INT Crea PI_SSE IN IN IN</tyj 	pe> BUF(*) EGER COUNT, DATATYP ates a persistent comm END_INIT(buf, count, c buf count datatype	 DEST, TAG, COMM, REQUEST, IERROR numication request for a buffered mode send. datatype, dest, tag, comm, request) initial address of send buffer (choice) number of elements sent (non-negative integer) type of each element (handle) 	
<tyj INT Crea PI_SSE IN IN IN IN</tyj 	pe> BUF(*) EGER COUNT, DATATYF ates a persistent comm END_INIT(buf, count, c buf count datatype dest	PE, DEST, TAG, COMM, REQUEST, IERROR nunication request for a buffered mode send. Hatatype, dest, tag, comm, request) initial address of send buffer (choice) number of elements sent (non-negative integer) type of each element (handle) rank of destination (integer)	
<tyj INT Crea PI_SSE IN IN IN IN IN IN</tyj 	pe> BUF(*) EGER COUNT, DATATYP ates a persistent comm END_INIT(buf, count, c buf count datatype dest tag comm	PE, DEST, TAG, COMM, REQUEST, IERROR nunication request for a buffered mode send. Hatatype, dest, tag, comm, request) initial address of send buffer (choice) number of elements sent (non-negative integer) type of each element (handle) rank of destination (integer) message tag (integer) communicator (handle)	
<tyj INT Crea PI_SSE IN IN IN IN</tyj 	pe> BUF(*) EGER COUNT, DATATYF ates a persistent comm END_INIT(buf, count, c buf count datatype dest tag	PE, DEST, TAG, COMM, REQUEST, IERROR nunication request for a buffered mode send. Hatatype, dest, tag, comm, request) initial address of send buffer (choice) number of elements sent (non-negative integer) type of each element (handle) rank of destination (integer) message tag (integer)	
<tyj INT Crea PI_SSE IN IN IN IN IN IN OUT</tyj 	pe> BUF(*) EGER COUNT, DATATYP ates a persistent comm END_INIT(buf, count, c buf count datatype dest tag comm request	PE, DEST, TAG, COMM, REQUEST, IERROR nunication request for a buffered mode send. Hatatype, dest, tag, comm, request) initial address of send buffer (choice) number of elements sent (non-negative integer) type of each element (handle) rank of destination (integer) message tag (integer) communicator (handle)	
<tyj INT Crea PI_SSE IN IN IN IN IN OUT bindi</tyj 	<pre>pe> BUF(*) EGER COUNT, DATATYF ates a persistent comm END_INIT(buf, count, c buf count datatype dest tag comm request ng _Ssend_init(const v</pre>	<pre>PE, DEST, TAG, COMM, REQUEST, IERROR numication request for a buffered mode send. Hatatype, dest, tag, comm, request) initial address of send buffer (choice) number of elements sent (non-negative integer) type of each element (handle) rank of destination (integer) message tag (integer) communicator (handle) communication request (handle)</pre>	
<tyj INT Crea IPI_SSE IN IN IN IN IN IN OUT S bindi</tyj 	<pre>pe> BUF(*) EGER COUNT, DATATYF ates a persistent comm END_INIT(buf, count, c buf count datatype dest tag comm request ng _Ssend_init(const v</pre>	PE, DEST, TAG, COMM, REQUEST, IERROR numication request for a buffered mode send. Hatatype, dest, tag, comm, request) initial address of send buffer (choice) number of elements sent (non-negative integer) type of each element (handle) rank of destination (integer) message tag (integer) communicator (handle) communication request (handle)	

Unofficial Draft for Comment Only

1 2 3 4 5 6 7 8 9 10 11 12 13	INTEC TYPE TYPE INTEC Fortran I MPI_SSENI <type INTEC</type 	GER, INTENT(IN) :: count, (MPI_Datatype), INTENT(IN (MPI_Comm), INTENT(IN) :: (MPI_Request), INTENT(OUT GER, OPTIONAL, INTENT(OUT binding D_INIT(BUF, COUNT, DATATY e> BUF(*) GER COUNT, DATATYPE, DEST) :: datatype comm) :: request
14 15	MPI_RSE	ND_INIT(buf, count, datatype,	dest, tag, comm, request)
16	IN	buf	initial address of send buffer (choice)
17 18	IN	count	number of elements sent (non-negative integer)
19	IN	datatype	type of each element (handle)
20	IN	dest	rank of destination (integer)
21 22	IN	tag	message tag (integer)
23	IN	comm	communicator (handle)
24 25	OUT	request	communication request (handle)
26 27	C bindin int MPI_H	0	f, int count, MPI_Datatype datatype,
28 29		int dest, int tag, M	PI_Comm comm, MPI_Request *request)
30	Fortran 2	2008 binding	
31			pe, dest, tag, comm, request, ierror)
32 33		(*), DIMENSION(), INTEN GER, INTENT(IN) :: count,	T(IN), ASYNCHRONOUS :: buf dest. tag
34		(MPI_Datatype), INTENT(IN	
35		(MPI_Comm), INTENT(IN) ::	
36		(MPI_Request), INTENT(OUT GER, OPTIONAL, INTENT(OUT	-
37 38			
39	Fortran l		
40		e> BUF(*)	PE, DEST, TAG, COMM, REQUEST, IERROR)
41 42	• 1		, TAG, COMM, REQUEST, IERROR
43	Creat	es a persistent communication	n object for a ready mode send operation.
44		-	
45			
46 47			
48			

MPI_REC	/_INIT(buf, count,	datatype, source, tag, comm, request)	1
OUT	buf	initial address of receive buffer (choice)	2 3
IN	count	number of elements received (non-negative integer)	3
IN	datatype	type of each element (handle)	5
IN	source	rank of source or MPI_ANY_SOURCE (integer)	6
IN	tag	message tag or MPI_ANY_TAG (integer)	7 8
IN	comm	communicator (handle)	9
OUT	request	communication request (handle)	10
	,	- · · ·	11 12
C bindin	g		12
int MPI_F		<pre>wbuf, int count, MPI_Datatype datatype, int source,</pre>	14
	int tag, M	PI_Comm comm, MPI_Request *request)	15
	2008 binding		16 17
		t, datatype, source, tag, comm, request, ierror)	18
), ASYNCHRONOUS :: buf :: count, source, tag	19
		INTENT(IN) :: datatype	20
	(MPI_Comm), INTE		21
TYPE	(MPI_Request), I	INTENT(OUT) :: request	22
INTEC	GER, OPTIONAL, I	INTENT(OUT) :: ierror	23
Fortran l	oinding		24 25
		T, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)	25 26
<type< td=""><td>e> BUF(*)</td><td></td><td>27</td></type<>	e> BUF(*)		27
INTEC	GER COUNT, DATAT	TYPE, SOURCE, TAG, COMM, REQUEST, IERROR	28
Creat	es a persistent con	nmunication request for a receive operation. The argument buf	29
	-	e user gives permission to write on the receive buffer by passing	30
the argum	ent to MPI_RECV.	_INIT.	31
-		ation request is inactive after it was created — no active com-	32 33
	n is attached to th		33 34
		uses a persistent request is initiated by the function	35
MPI_STAF	XI.		36
			37
MPI_STAF	RT(request)		38
INOUT	request	communication request (handle)	39
	•		40 41
C bindin			41
int MPI_S	Start(MPI_Reques	st *request)	43
Fortran 2	2008 binding		44
	c(request, ierro	or)	45
TYPE	(MPI_Request), I	INTENT(INOUT) :: request	46
INTEC	GER, OPTIONAL, I	INTENT(OUT) :: ierror	47

1 2 3		Dinding F(REQUEST, IERROR) GER REQUEST, IERROR	
4 5 7 8 9 10 11 12 13 14 15	associated If the before the and until to The c described MPI_SENE call to MF	request should be inactive request is for a send with call is made. The commu- the operation completes. all is local, with similar se in Section 3.7. That is, a D_INIT starts a communic	ndle returned by one of the previous five calls. The e. The request becomes active once the call is made. ready mode, then a matching receive should be posted unication buffer should not be modified after the call, mantics to the nonblocking communication operations call to MPI_START with a request created by ation in the same manner as a call to MPI_ISEND; a created by MPI_BSEND_INIT starts a communication _IBSEND; and so on.
16 17		RTALL(count, array_of_req	
18	IN	count	list length (non-negative integer)
19 20	INOUT	array_of_requests	array of requests (array of handles)
21	C binding	g	
22	int MPI_S	Startall(int count, MP)	[_Request array_of_requests[])
23 24	Fortran 2	2008 binding	
24 25		call(count, array_of_r	equests, ierror)
26	INTEC	GER, INTENT(IN) :: cour	it it
27 28		(MPI_Request), INTENT() GER, OPTIONAL, INTENT()	INOUT) :: array_of_requests(count) DUT) :: ierror
29	Fortran k	ainding	
30 31	MPI_START	TALL(COUNT, ARRAY_OF_RI GER COUNT, ARRAY_OF_RE	
32 33	Start	all communications assoc	niated with requests in array_of_requests. A call to
34			uests) has the same effect as calls to
35			executed for $i=0$,, count-1, in some arbitrary order.
36			a call to MPI_START or MPI_STARTALL is completed
37	•		F, or one of the derived functions described in Sec-
38		-	ctive after successful completion of such call. The re- activated anew by an MPI_START or MPI_STARTALL
$\frac{39}{40}$	call.	of deallocated and it can be	activated allew by all WIT_START of WITT_STARTALE
40 41		sistent request is deallocat	ed by a call to MPI_REQUEST_FREE (Section 3.7.3).
42	-	_	E can occur at any point in the program after the per-
43	sistent req	uest was created. However	, the request will be deallocated only after it becomes
44		-	uld not be freed. Otherwise, it will not be possible to
45		_	Collective operation requests (defined in Section 6.12
46		_	ective operations, and Section 6.13 and Section 8.8 for
47	persistent	collective operations) mus	t not be freed while active. It is preferable, in general,
48			

to free requests when they are inactive. If this rule is followed, then the functions described in this section will be invoked in a sequence of the form,

Create (Start Complete)* Free

where * indicates zero or more repetitions. If the same communication object is used in several concurrent threads, it is the user's responsibility to coordinate calls so that the correct sequence is obeyed.

A send operation initiated with MPI_START can be matched with any receive operation and, likewise, a receive operation initiated with MPI_START can receive messages generated by any send operation.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 19.1.10–19.1.20. (End of advice to users.)

3.10 Null Processes

In many instances, it is convenient to specify a "dummy" source or destination for communication. This simplifies the code that is needed for dealing with boundaries, for example, in the case of a non-circular shift done with calls to send-receive.

The special value MPI_PROC_NULL can be used instead of a rank wherever a source or a destination argument is required in a call. A communication with process MPI_PROC_NULL has no effect. A send to MPI_PROC_NULL succeeds and returns as soon as possible. A receive from MPI_PROC_NULL succeeds and returns as soon as possible with no modifications to the receive buffer. When a receive with source = MPI_PROC_NULL is executed then the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG and count = 0. A probe or matching probe with source = MPI_PROC_NULL succeeds and returns as soon as possible, and the status object returns source = MPI_PROC_NULL succeeds and returns as soon as possible, and the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG and count = 0. A matching probe (cf. Section 3.8.2) with source = MPI_PROC_NULL returns flag = true, message = MPI_MESSAGE_NO_PROC, and the status object returns source = MPI_PROC_NULL, tag = MPI_PROC_NULL returns flag = MPI_ANY_TAG, and count = 0.

 $\mathbf{2}$

 24



Chapter 4

Partitioned Point-to-Point Communication

4.1 Introduction

Partitioned communication extends persistent point-to-point communication as defined in Chapter 3. Partitioned communication operations are matched based on the order in which the local initialization calls are performed. Partitioned communication is "partitioned" because it allows for multiple contributions of data to be made, potentially, from multiple actors (e.g., threads or tasks) in an MPI process to a single communication operation.

Advice to users. The techniques of partitioned communication were known as "finepoints" before their adoption into the MPI standard. We refer the interested reader to the original literature describing the design goals, functioning, initial implementation and performance improvements [28, 29]. (End of advice to users.)

Partitioned communication operations use a persistent communication style that involves a sequence of start and test or wait operations. For this sequence partitioned communications use MPI_START or MPI_STARTALL calls and completion mechanisms (MPI_TEST or MPI_WAIT). Partitioned communication is different in three fundamental ways from persistent point-to-point operations in MPI. First, partitioned communication allows additional partitioned test function calls that can expose partial completion of the operation. Second, partitioned communication may perform all of the initialization required to enable data transfer as early as its initialization phase. Third, partitioned communication allows for MPI to be independently notified of multiple contributions from the send-side to a single data buffer of a single MPI message.

39 The rationale behind having different initialization behavior allowed Rationale. for partitioned communication as opposed to persistent point-to-point is to enable 40 41 flexibility and optimization possibilities in implementations. Buffer setup can occur in 42the partitioned communication initialization functions (see Section 4.2.1). However, such negotiation can be deferred until data is to be moved between two processes. 4344This means that partitioned communication can lazily negotiate as late as testing for completion of the operation on the first iteration of a partitioned communication 4546start and test or wait operations. Matching still occurs as if matching happened 47at the partitioned communication initialization functions as noted in the function 48 descriptions. (End of rationale.)

Unofficial Draft for Comment Only

9

10 11

17 18

19

20

21

22

23

 24

25 26

27 28

29

30

31

32

33

34

35

36

37

CHAPTER 4. PARTITIONED POINT-TO-POINT COMMUNICATION

1 2

4

94

4.2 Semantics of Partitioned Point-to-Point Communication

MPI guarantees certain general properties of partitioned point-to-point communication 3 progress, which are described in this section.

Persistent communications use opaque MPI_REQUEST objects as described in Sec-5tion 3. Partitioned communication uses these same semantics for MPI_REQUEST objects. 6

Partitioned communication provides fine-grained transfers on either or both sides of a 7 8 send-receive operation described by requests. Persistent communication semantics are ideal for partitioned communication: they provide MPI_PSEND_INIT and MPI_PRECV_INIT 9 functions that allow partitioned communication setup to occur prior to message transfers. 10 Partitioned communication initialization functions are local. The partitioned communica-11 tion initialization includes inputs on the number of user-visible partitions on the send-side 12and receive-sides, which may differ. Valid partitioned communication operations must have 13 one or more partitions specified. 14

Once an MPI_PSEND_INIT call has been made, the user may start the operation with 15a call to a starting procedure and complete the operation with a number of MPI_PREADY 16calls equal to the requested number of send partitions followed by a call to a completing 17procedure. A call to MPI_PREADY notifies the MPI library that a specified portion of the 18 data buffer (a specific partition) is ready to be sent. Notification of partial completion can 19 be done via fine-grained MPI_PARRIVED calls at the receiver before a final MPI_TEST/ 20

MPI_WAIT on the request itself; the latter represents overall operation completion upon 21success. A full set of methods for starting and completing partitioned communication is 22 given in the following sections. 23

 24 25

26

27

 31

32

33 34

35

36

37

38 39

40

41 42

Advice to users. Having a large number of receiver-side partitions can increase overheads as the completion mechanism may need to work with finer-grained notifications. Using a small number of receiver-side partitions may provide higher performance.

A large number of sender-side partitions may be aggregated by an MPI implementa-2829 tion, making performance concerns of a large number of sender-side partitions poten-30 tially less impactful than receiver-side granularity. (End of advice to users.)

Advice to implementors. It is expected that an MPI implementation will attempt to balance latency and aggregation for data transfers for the requested partition counts on the sender-side and receiver-side to allow optimization for different hardware. A high quality implementation may perform significant optimizations to enhance performance in this way; they may, for example, resize the data transfers of the partitions to combine partitions in fractional partition sizes (e.g., 2.5 partitions in a single data transfer). (End of advice to implementors.)

Example 4.1 shows a simple partitioned transfer in which the sender-side and receiverside partitioning is identical in partition count.

Example 4.1

- 43 44
- 45
- 46
- 4748

```
1
#include "mpi.h"
                                                                                      2
#define PARTITIONS 8
                                                                                      3
#define COUNT 5
int main(int argc, char *argv[])
                                                                                      4
ſ
                                                                                      5
  double message[PARTITIONS*COUNT];
                                                                                      6
 MPI_Count partitions = PARTITIONS,
  int source = 0, dest = 1, tag = 1, flag = 0;
  int myrank, i;
                                                                                      10
  int provided;
                                                                                      11
  MPI_Request request;
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
                                                                                      12
  if (provided < MPI_THREAD_SERIALIZED) MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE) 3
                                                                                      14
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
                                                                                      15
  if (myrank == 0)
                                                                                      16
  {
                                                                                      17
     MPI_Psend_init(message, partitions, COUNT, MPI_DOUBLE, dest, tag,
                                                                                      18
                MPI_COMM_WORLD, MPI_INFO_NULL, &request);
                                                                                      19
     MPI_Start(&request);
     for(i = 0; i < partitions; ++i)</pre>
                                                                                      20
                                                                                      21
     {
        /* compute and fill partition #i, then mark ready: */
                                                                                      22
                                                                                      23
        MPI_Pready(i, request);
                                                                                      ^{24}
     }
                                                                                      25
     while(!flag)
                                                                                      26
     {
        /* do useful work #1 */
                                                                                      27
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
                                                                                      28
                                                                                      29
        /* do useful work #2 */
                                                                                      30
     }
                                                                                      31
     MPI_Request_free(&request);
  }
                                                                                      32
                                                                                      33
  else if (myrank == 1)
                                                                                      34
  £
     MPI_Precv_init(message, partitions, COUNT, MPI_DOUBLE, source, tag,
                                                                                      35
                MPI_COMM_WORLD, MPI_INFO_NULL, &request);
                                                                                      36
                                                                                      37
     MPI_Start(&request);
     while(!flag)
                                                                                      38
                                                                                      39
     {
        /* do useful work #1 */
                                                                                      40
                                                                                      41
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
                                                                                      42
        /* do useful work #2 */
     }
                                                                                      43
                                                                                      44
     MPI_Request_free(&request);
  }
                                                                                      45
                                                                                      46
 MPI_Finalize();
                                                                                      47
  return 0;
                                                                                      48
}
```

Rationale. Partitioned communication is designed to provide opportunities for MPI implementations to optimize data transfers. MPI is free to choose how many transfers to do within a partitioned communication send independent of how many partitions are reported as ready to MPI through MPI_PREADY calls. Aggregation of partitions is permitted but not required. Ordering of partitions is permitted but not required. A naive implementation can simply wait for the entire message buffer to be marked ready before any transfer(s) occur and could wait until the completion function is called on a request before transferring data. However, this modality of communication gives MPI implementations far more flexibility in data movement than non-partitioned communications. (*End of rationale.*)

4.2.1 Communication Initialization and Starting with Partitioning

Initialization of partitioned communication operations use the initialization calls described below. Subsequent to initialization, MPI_START/MPI_STARTALL are used as the first indication to MPI that a message transfer will occur. For send-side operations, neither initializing nor starting the operation enables transfer of any part of the user buffer. Freeing or canceling a partitioned communication request that is active (i.e., initialized and started) and not completed is erroneous. After the partitioned communication operation is started, individual partitions of a message are indicated as ready to be sent by MPI via the MPI_PREADY function, described below.

 24

MPI_PSEND_INIT(but,	partitions, cour	nt, datatype, c	lest, tag, o	comm, into,	request)

24	IN	buf	initial address of send buffer (choice) (choice)
26	IN	partitions	number of partitions (non-negative integer)
27 28 29	IN	count	number of elements send per partition (non-negative integer)
30	IN	datatype	type of each element (handle)
31	IN	dest	rank of destination (integer)
32	IN	tag	message tag (integer)
33 34	IN	comm	communicator (handle)
35	IN	info	info argument (handle)
36	Ουτ	request	communication request (handle)
37 38	C hindin		• · · /
39	C binding	•	partitions, MPI_Count count,
40	1110 111 1_1	₩ ₩	pe, int dest, int tag, MPI_Comm comm,
41 42		MPI_Info info, MPI_R	0
43	Fortran 2	2008 binding	
44		0	ount, datatype, dest, tag, comm, info,
45		request, ierror)	·····, ·····, ·····, ·····, ·····, ·····, ·····, ·····, ·····, ·····, ·····, ·····, ·····, ·····, ·····, ·····,
46	TYPE ((*), DIMENSION(), INTEN	T(IN) :: buf
47		ER, INTENT(IN) :: partit	
48		<pre>ER(KIND=MPI_COUNT_KIND),</pre>	-

```
TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_PSEND_INIT(BUF, PARTITIONS, COUNT, DATATYPE, DEST, TAG, COMM, INFO,
              REQUEST, IERROR)
    <type> BUF(*)
    INTEGER PARTITIONS, DATATYPE, DEST, TAG, COMM, INFO, REQUEST, IERROR
    INTEGER(KIND=MPI_COUNT_KIND) COUNT
MPI_PSEND_INIT creates a partitioned communication request and binds to it all the ar-
guments of a partitioned send operation. Matching follows the same MPI matching rules
as for point-to-point communication (see Chapter 3) with communicator, tag and source
dictating message matching. In the event that the communicator, tag and source do not
uniquely identify a message, the order in which partitioned communication initialization
calls are made is the order in which they will eventually match. This operation can only
match with partitioned communication initialization operations, therefore it is required to
be matched with a corresponding MPI_PRECV_INIT call. Partitioned communication ini-
tialization calls are local. It is erroneous to provide a partitions value \leq 0. Send-side and
receive-side buffers must be identical in size.
     Advice to implementors. Unlike MPI_SEND_INIT, MPI_PSEND_INIT can be matched
     as early as the initialization call. Also, unlike MPI_SEND_INIT, MPI_PSEND_INIT
     takes an info argument. (End of advice to implementors.)
```

	N/ INUT(huf postitions count	detection dest tag community	29
MPI_PREC	v_INTT(but, partitions, count,	datatype, dest, tag, comm, info, request)	30
IN	buf	initial address of recv buffer (choice) (choice)	31
IN	partitions	number of partitions (non-negative integer)	32
IN	count	number of elements send per partition (non-negative	33 34
		integer)	34 35
IN	datatype	type of each element (handle)	36
IN	dest	rank of destination (integer)	37
IN	tag	message tag (integer)	38 39
IN	comm	communicator (handle)	40
IN	info	info argument (handle)	41
		0 ()	42
OUT	request	communication request (handle)	43
			44
C binding	5		45

int MPI_Precv_init(void *buf, int partitions, MPI_Count count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Info info, MPI_Request *request) 48

Unofficial Draft for Comment Only

97

1

 $\mathbf{2}$

3

4

5 6

8

9

10

11

12 13

14

15

16

17

18

19

20

21

22

23

 24

25

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Precv_init(buf, partitions, count, datatype, dest, tag, comm, info,
3
                    request, ierror)
4
          TYPE(*), DIMENSION(...), INTENT(IN) :: buf
5
          INTEGER, INTENT(IN) :: partitions, dest, tag
6
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
7
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
          TYPE(MPI_Comm), INTENT(IN) :: comm
9
          TYPE(MPI_Info), INTENT(IN) :: info
10
          TYPE(MPI_Request), INTENT(OUT) :: request
11
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     Fortran binding
13
     MPI_PRECV_INIT(BUF, PARTITIONS, COUNT, DATATYPE, DEST, TAG, COMM, INFO,
14
                    REQUEST, IERROR)
15
          <type> BUF(*)
16
          INTEGER PARTITIONS, DATATYPE, DEST, TAG, COMM, INFO, REQUEST,
                                                                                 IERROR
17
          INTEGER(KIND=MPI_COUNT_KIND) COUNT
18
19
           Rationale. The info argument is provided in order to support per-operation implementation-
20
           defined info keys. (End of rationale.)
21
22
          MPI_PRECV_INIT creates a partitioned communication receive request and binds to it
23
     all the arguments of a partitioned receive operation. This operation can only match with
^{24}
     partitioned communication initialization operations, therefore the MPI library is required to
25
     match MPI_PRECV_INIT calls only with a corresponding MPI_PSEND_INIT call. Matching
26
     follows the same MPI matching rules as for point-to-point communication (see Chapter 3)
27
     with communicator, tag and source dictating message matching. In the event that the
28
     communicator, tag and source do not uniquely identify a message, the order in which
29
     partitioned communication initialization calls are made is the order in which they will
30
     eventually match. Partitioned communication initialization calls are local. That is,
^{31}
     MPI_PRECV_INIT may return before the operation completes. It is erroneous to provide a
32
     partitions value \leq 0. Wildcards for source and tag are not allowed.
33
34
           Advice to implementors. Unlike MPI_RECV_INIT, MPI_PRECV_INIT may communi-
35
           cate. Also unlike MPI_RECV_INIT, MPI_PRECV_INIT takes an info argument. (End
36
           of advice to implementors.)
37
38
39
40
     MPI_PREADY(partition, request)
41
       IN
                 partition
                                             partition to mark ready for transfer (non-negative
42
                                             integer)
43
44
       INOUT
                 request
                                             partitioned communication request (handle)
45
46
     C binding
47
     int MPI_Pready(int partition, MPI_Request *request)
48
```

```
1
Fortran 2008 binding
                                                                                           \mathbf{2}
MPI_Pready(partition, request, ierror)
                                                                                           3
    INTEGER, INTENT(IN) :: partition
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                           4
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                           5
                                                                                           6
Fortran binding
MPI_PREADY(PARTITION, REQUEST, IERROR)
    INTEGER PARTITION, REQUEST, IERROR
                                                                                           9
                                                                                           10
    MPI_PREADY is a send-side call that indicates that a given partition is ready to be
                                                                                           11
transferred. It is erroneous to use MPI_PREADY on any request object that does not
correspond to a partitioned send operation. The partitioning is defined by the
                                                                                           12
                                                                                           13
MPI_PSEND_INIT call. Partition numbering starts at zero and ranges to one less than
                                                                                           14
the number of partitions declared in the MPI_PSEND_INIT call. Specifying a partition
                                                                                           15
number that is equal to or larger than the number of partitions is erroneous. After a call
                                                                                           16
to MPI_START/MPI_STARTALL, all partitions associated with that operation are inactive.
                                                                                           17
A call to MPI_PREADY marks the indicated partition as active. Calling MPI_PREADY on
                                                                                           18
an active partition is erroneous.
                                                                                           19
                                                                                           20
MPI_PREADY_RANGE(partition_low, partition_high, request)
                                                                                          21
                                                                                           22
  IN
            partition_low
                                        partition to mark lowest partition ready for transfer
                                                                                           23
                                        (non-negative integer)
                                                                                           ^{24}
  IN
            partition_high
                                        partition to mark highest partition ready for transfer
                                                                                           25
                                        (non-negative integer)
                                                                                           26
  INOUT
            request
                                        partitioned communication request (handle)
                                                                                           27
                                                                                           28
C binding
                                                                                           29
int MPI_Pready_range(int partition_low, int partition_high,
                                                                                           30
               MPI_Request *request)
                                                                                           31
                                                                                           32
Fortran 2008 binding
                                                                                           33
MPI_Pready_range(partition_low, partition_high, request, ierror)
                                                                                           34
    INTEGER, INTENT(IN) :: partition_low, partition_high
                                                                                           35
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                           36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                           37
                                                                                           38
Fortran binding
                                                                                           39
MPI_PREADY_RANGE(PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR)
                                                                                           40
    INTEGER PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR
                                                                                           41
    A call to MPI_PREADY_RANGE has the same effect as calls to MPI_PREADY, executed
                                                                                           42
for i=partition_low ,..., partition_high, in some arbitrary order. Calls to
                                                                                           43
MPI_PREADY_RANGE follow the same rules as those for MPI_PREADY calls.
                                                                                           44
                                                                                           45
                                                                                           46
                                                                                           47
```

	100	UNAPIER 4. PAI	RITTIONED POINT-TO-POINT COMMUNICATION
1	MPI_PREA	ADY_LIST(length, array_o	f_partitions, request)
2 3	IN	length	list length (integer)
4	INOUT	array_of_partitions	array of partitions (array of non-negative integers)
5	INOUT	request	partitioned communication request (handle)
6		request	partitioned communication request (manalo)
7	C binding	.	
8		5	n, int array_of_partitions[],
9		MPI_Request *req	
10 11	Fortran 2	2008 binding	
12		0	_of_partitions, request, ierror)
13		ER, INTENT(IN) :: lei	
14			array_of_partitions(length)
15		MPI_Request), INTENT	
16	INTEG	ER, OPTIONAL, INTENT	(DUT) :: ierror
17	Fortran b	oinding	
18 19			_OF_PARTITIONS, REQUEST, IERROR)
20	INTEG	ER LENGTH, ARRAY_OF_I	PARTITIONS(*), REQUEST, IERROR
21	A cal	l to MPI_PREADY_LIST	has the same effect as calls to
22	MPI_PREA	ADY, executed for the pa	artitions specified in the range $array_of_partitions[0]$
23	-		1] of the array_of_partitions, executed in some arbitrary
24	order. Cal	ls to MPI_PREADY_LIST	follow the same rules as those for MPI_PREADY calls.
25	400 6		
26 27	4.2.2 Co	mmunication Completior	under Partitioning
28			TEST (and variants) are used to complete a partitioned
29		-	npletion of a partitioned send operation indicates that
30			START/MPI_STARTALL to restart the operation and
31	*	· · · ·	PREADY_RANGE or MPI_PREADY_LIST. Alternatively, ned communication request after the completion of the
32			ng process, completion of the partitioned send operation
33	-	· ·	s of the message have all been received.
34 35		-	ed receive operation through MPI_WAIT or MPI_TEST
36			ntains all of the partitions. A function for probing the
37			fer is provided by MPI_PARRIVED. The MPI_PARRIVED
38			if the message data for the indicated partition has been
39			Upon success, the receiver becomes free to access the others that previously completed for that operation).
40	mulcateu j	farthful (as well as ally (stillers that previously completed for that operation).
41			
42 43			
44			
45			
46			
47			
48			

CHAPTER 4. PARTITIONED POINT-TO-POINT COMMUNICATION

MPI_PARR	IVED(request, partition, flag)		1			
INOUT	request	partitioned communication request (handle)	2			
IN	•		3			
	partition	partition to be tested (non-negative integer)	4			
OUT	flag	true if operation completed on the specified partition,	5			
		false if not (boolean)	6			
			7 8			
C binding	-		9			
int MPI_P	arrived(MPI_Request *req	uest, int partition, int *flag)	10			
Fortran 2	008 binding		11			
MPI_Parrived(request, partition, flag, ierror) 12						
TYPE(TYPE(MPI_Request), INTENT(INOUT) :: request					
INTEGER, INTENT(IN) :: partition 14						
	AL, INTENT(OUT) :: flag		15			
INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	16			
Fortran b	inding		17			
MPI_PARRI	VED(REQUEST, PARTITION, 1	FLAG, IERROR)	18			
INTEG	ER REQUEST, PARTITION, I	ERROR	19			
LOGIC	AL FLAG		20 21			
The fr	unction MPI PARRIVED can	be used to test partial completion of partitioned	21			
		RIVED on an active partitioned communications	23			
-		tion identified by request for the specified	24			
		ot marked as complete/inactive by this operation.	25			
		peration is required to complete the message, as	26			
described i	n Chapter 3. MPI_PARRIV	ED may be called multiple times for a partition.	27			
MPI_PARR	IVED may be called with a	null or inactive request argument. In either case,	28			
		Calling MPI_PARRIVED on a request that does not	29			
correspond	to a partitioned receive oper	ation is erroneous.	30			
			31			
4.2.3 Sen	nantics of Communications i	n Partitioned Mode	32			
The seman	tics of nonblocking partitione	d communication are defined by suitably extending	33			
			34			

the definitions in Section 3.5.

Interpretation of count and datatype for partitioned communication Partitioned communica-37 tion uses the count and datatype arguments in the partitioned communication initialization 38 functions to describe a single partition. The argument partitions specifies how many equal 39partitions of a number (count) of datatypes make up the entire buffer to be transferred in the partitioned communication. As partitioned communication describes many partitions, 41 using absolute displacements in datatypes (e.g., MPI_BOTTOM) is not supported. Partitions 42are contiguous in memory, there is no padding in between partitions. Once a partitioned send operation is started, each partition must be marked as ready using MPI_PREADY and the operation must be completed using a completion function, such as MPI_TEST or MPI_WAIT.

35 36

40

43

44

45

⁴⁶ 47

Order Matching follows the same MPI matching rules as for point-to-point communication (see Chapter 3) with communicator, tag and source dictating message matching. In the event that the communicator, tag and source do not uniquely identify the message, the order in which partitioned communication initialization calls are made is the order in which they will eventually match.

4.3 Partitioned Communication Examples

This section provides concrete examples of the utility of partitioned communication in realistic settings.

```
4.3.1 Partition Communication with Threads/Tasks Using OpenMP 4.0 and greater
```

The equal partitioning on send-side and receive-side in Example 4.1 is shown using threads. In this case, the receive-side uses the same number of partitions as the sending-side like the previous example, but this example uses multiple threads on the sending-side. Note that the MPI_PSEND_INIT and MPI_PRECV_INIT functions match each other like in the previous example.

```
<sup>20</sup> Example 4.2
```

```
21
     #include "mpi.h"
22
     #define NUM_THREADS 8
23
     #define PARTITIONS 8
24
     #define PARTLENGTH 16
25
     int main( int argc, char *argv[]) /* same send/recv partitioning */
26
     {
27
       double message[PARTITIONS*PARTLENGTH];
28
       int partitions = PARTITIONS;
29
       int partlength = PARTLENGTH;
30
       int count = 1, source = 0, dest = 1, tag = 1, flag = 0;
31
       int myrank;
32
       int provided;
33
       MPI_Request request;
34
       MPI_Info info = MPI_INFO_NULL;
35
       MPI_Datatype xfer_type;
36
       MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
37
       if (provided < MPI_THREAD_SERIALIZED) MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);</pre>
38
       MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
39
       MPI_Type_contiguous(partlength, MPI_DOUBLE, &xfer_type);
40
       MPI_Type_commit(&xfer_type);
41
                            /* code for process zero */
       if (myrank == 0)
42
       {
43
          MPI_Psend_init(message, partitions, count, xfer_type, dest, tag,
44
               info, MPI_COMM_WORLD, &request);
45
          MPI_Start(&request);
46
47
          #pragma omp parallel for shared(request) num_threads(NUM_THREADS)
48
```

6 7

8 9

10

11 12

13

```
1
     for (int i=0; i<partitions; i++)</pre>
                                                                                          \mathbf{2}
     {
                                                                                          3
         /* compute and fill partition #i, then mark ready: */
        MPI_Pready(i, request);
                                                                                          4
     }
                                                                                          5
     while(!flag)
                                                                                          6
     {
         /* Do useful work */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
                                                                                          9
                                                                                          10
         /* Do useful work */
     }
                                                                                          11
     MPI_Request_free(&request);
                                                                                          12
  }
                                                                                          13
                                                                                          14
  else if (myrank == 1) /* code for process one */
                                                                                          15
  {
                                                                                          16
     MPI_Precv_init(message, partitions, count, xfer_type, source, tag,
                                                                                          17
            info, MPI_COMM_WORLD, &request);
                                                                                          18
     MPI_Start(&request);
                                                                                          19
     while(!flag)
     ſ
                                                                                          20
                                                                                          21
         /* Do useful work */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
                                                                                          22
                                                                                          23
         /* Do useful work */
     }
                                                                                          ^{24}
                                                                                          25
     MPI_Request_free(&request);
                                                                                          26
  }
  MPI_Finalize();
                                                                                          27
                                                                                          28
  return 0;
                                                                                          29
}
                                                                                          30
                                                                                          31
       Send-only Partitioning Example with Tasks and OpenMP version 4.0 and greater
4.3.2
                                                                                          32
The previous example is tailored specifically for send-side partitioning using threads. This
                                                                                          33
is an example where parallel task producers produce input to part of an overall buffer; they
                                                                                          34
complete in any order and contribute to the overall buffer.
                                                                                          35
                                                                                          36
Example 4.3
                                                                                          37
                                                                                          38
#include "mpi.h"
#define NUM_THREADS 8
                                                                                          39
#define NUM_TASKS 64
                                                                                          40
                                                                                          41
#define PARTITIONS NUM_TASKS
                                                                                          42
#define PARTLENGTH 16
#define MESSAGE_LENGTH PARTITIONS*PARTLENGTH
                                                                                          43
                                                                                          44
int main( int argc, char *argv[]) /* send-side partitioning */
```

double message [MESSAGE_LENGTH];

int send_partitions = PARTITIONS,

send_partlength = PARTLENGTH,

45

46

47

104

```
1
           recv_partitions = 1,
2
           recv_partlength = PARTITIONS*PARTLENGTH;
3
       int count = 1, source = 0, dest = 1, tag = 1, flag = 0;
4
       int myrank;
5
       int provided;
6
       MPI_Request request;
7
       MPI_Info info = MPI_INFO_NULL;
8
       MPI_Datatype send_type;
9
       MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
10
       if (provided < MPI_THREAD_SERIALIZED) MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
11
       MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
       MPI_Type_contiguous(send_partlength, MPI_DOUBLE, &send_type);
12
13
       MPI_Type_commit(&send_type);
14
15
                           /* code for process zero */
       if (myrank == 0)
16
       {
17
          MPI_Psend_init(message, send_partitions, count, send_type, dest, tag,
18
                     info, MPI_COMM_WORLD, &request);
19
          MPI_Start(&request);
20
21
          #pragma omp parallel shared(request) num_threads(NUM_THREADS)
22
          Ł
23
             #pragma omp single
24
             {
25
               /* single thread creates 64 tasks to be executed by 8 threads */
26
               for (int partition_num=0; partition_num<NUM_TASKS; partition_num++)
27
               {
28
                  #pragma omp task firstprivate(partition_num)
29
                   {
30
                    /* compute and fill partition #partition_num, then mark ready: */
31
                   /* buffer is filled in arbitrary order from each task */
32
                   MPI_Pready(partition_num, request);
33
                   } /*end task*/
34
               } /* end for */
35
             } /* end single */
36
          } /* end parallel */
37
          while(!flag)
38
          {
39
             /* Do useful work */
40
             MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
41
             /* Do useful work */
42
          }
43
          MPI_Request_free(&request);
44
       }
45
       else if (myrank == 1) /* code for process one */
46
       ſ
47
          MPI_Precv_init(message, recv_partitions, recv_partlength, MPI_DOUBLE,
48
                     source, tag, info, MPI_COMM_WORLD, &request);
```

Unofficial Draft for Comment Only

```
MPI_Start(&request);
while(!flag)
{
    /* Do useful work */
    MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
    /* Do useful work */
    }
    MPI_Request_free(&request);
}
MPI_Finalize();
return 0;
}
```

4.3.3 Send and Receive Partitioning Example with OpenMP version 4.0 and greater

This example demonstrates receive-side partial completion notification using more than one partition per receive-side thread. It uses a naive flag based method to test for multiple completed partitions per thread. Note that this means that some threads may be busy polling for completion of assigned partitions when partitions are available to work on that were not assigned to the polling threads in this example. More advanced work stealing methods could be employed for greater efficiency. Like previous examples, it also demonstrates send-side production of input to part of an overall buffer. This example also uses different send-side and receive-side partitioning.

Example 4.4

```
27
#include "mpi.h"
                                                                                      28
#define NUM_THREADS 64
                                                                                      29
#define PARTITIONS NUM_THREADS
#define PARTLENGTH 16
                                                                                      30
#define MESSAGE_LENGTH PARTITIONS*PARTLENGTH
                                                                                      31
int main( int argc, char *argv[]) /* send-side partitioning */
                                                                                      32
                                                                                      33
{
                                                                                      34
  double message [MESSAGE_LENGTH];
                                                                                      35
  int send_partitions = PARTITIONS,
                                                                                      36
      send_partlength = PARTLENGTH,
                                                                                      37
      recv_partitions = PARTITIONS*2,
                                                                                      38
      recv_partlength = PARTLENGTH/2;
                                                                                      39
  int source = 0, dest = 1, tag = 1,
                                                                                      40
  flag = 0;
                                                                                      41
  int myrank;
                                                                                      42
  int provided;
  MPI_Request request;
                                                                                      43
                                                                                      44
  MPI_Info info = MPI_INFO_NULL;
                                                                                      45
  MPI_Datatype send_type;
                                                                                      46
  MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
  if (provided < MPI_THREAD_SERIALIZED) MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE)<sup>47</sup>
                                                                                      48
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
```

```
1
       MPI_Type_contiguous(send_partlength, MPI_DOUBLE, &send_type);
2
       MPI_Type_commit(&send_type);
3
4
       if (myrank == 0)
                             /* code for process zero */
5
       ſ
6
          MPI_Psend_init(message, send_partitions, 1, send_type, dest, tag,
7
                     info, MPI_COMM_WORLD, &request);
8
          MPI_Start(&request);
9
          #pragma omp parallel for shared(request) num_threads(NUM_THREADS)
10
          for (int i=0; i<partitions; i++)</pre>
11
          {
12
              /* compute and fill partition #i, then mark ready: */
13
             MPI_Pready(i, request);
14
          }
15
          while(!flag)
16
          ſ
17
             /* Do useful work */
18
             MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
19
              /* Do useful work */
20
          }
21
          MPI_Request_free(&request);
22
       }
23
       else if (myrank == 1) /* code for process one */
24
       {
25
          MPI_Precv_init(message, recv_partitions, recv_partlength, MPI_DOUBLE,
26
                     source, tag, info, MPI_COMM_WORLD, &request);
27
          MPI_Start(&request);
28
          #pragma omp parallel for shared(request) num_threads(NUM_THREADS)
29
          for (int j=0; j<recv_partitions; j+=2)</pre>
30
          {
31
              int part1_complete = 0;
32
              int part2_complete = 0;
33
              while(part1_complete == 0 || part2_complete == 0)
34
              {
                 /* test partition #j and #j+1 */
35
36
                 MPI_Parrived(request, j, &flag);
37
                 if(flag && part1_complete == 0)
38
                 £
39
                    part1_complete++;
40
                    /* Do work using partition j data */
41
                 }
42
                 if (j+1 < recv_partitions) {</pre>
43
                   MPI_Parrived(request, j+1, &flag);
44
                   if(flag && part2_complete == 0)
45
                   {
46
                      part2_complete++;
47
                      /* Do work using partition j+1 */
48
                   }
```

```
}
                                                                                                      1
                                                                                                      \mathbf{2}
              else {
                                                                                                      3
                   part2_complete++;
                                                                                                      4
              }
                                                                                                      5
            }
          }
                                                                                                      6
      }
                                                                                                      7
      while(!flag)
                                                                                                      8
                                                                                                      9
      {
                                                                                                      10
          /* Do useful work */
         MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
                                                                                                      11
          /* Do useful work */
                                                                                                      12
      }
                                                                                                      13
      MPI_Request_free(&request);
                                                                                                      14
  }
                                                                                                      15
  MPI_Finalize();
                                                                                                      16
                                                                                                      17
  return 0;
                                                                                                      18
}
                                                                                                      19
                                                                                                     20
                                                                                                     21
                                                                                                     22
                                                                                                     23
                                                                                                      ^{24}
                                                                                                     25
                                                                                                      26
                                                                                                     27
                                                                                                     28
                                                                                                     29
                                                                                                     30
                                                                                                      ^{31}
                                                                                                     32
                                                                                                     33
                                                                                                     34
                                                                                                     35
                                                                                                     36
                                                                                                     37
                                                                                                     38
                                                                                                     39
                                                                                                      40
                                                                                                      41
                                                                                                     42
                                                                                                      43
                                                                                                      44
                                                                                                      45
                                                                                                      46
                                                                                                      47
                                                                                                      48
```



Chapter 5

Datatypes

Basic datatypes were introduced in Section 3.2.2 and in Section 3.3. In this chapter, this model is extended to describe any data layout. We consider general datatypes that allow one to transfer efficiently heterogeneous and noncontiguous data. We conclude with the description of calls for explicit packing and unpacking of messages.

5.1**Derived** Datatypes

Up to here, all point to point communications have involved only buffers containing a sequence of identical basic datatypes. This is too constraining on two accounts. One often wants to pass messages that contain values with different datatypes (e.g., an integer count, followed by a sequence of real numbers); and one often wants to send noncontiguous data (e.g., a sub-block of a matrix). One solution is to pack noncontiguous data into a contiguous buffer at the sender site and unpack it at the receiver site. This has the disadvantage of requiring additional memory-to-memory copy operations at both sites, even when the communication subsystem has scatter-gather capabilities. Instead, MPI provides mechanisms to specify more general, mixed, and noncontiguous communication buffers. It is up to the implementation to decide whether data should be first packed in a contiguous buffer before being transmitted, or whether it can be collected directly from where it resides.

The general mechanisms provided here allow one to transfer directly, without copying, objects of various shapes and sizes. It is not assumed that the MPI library is cognizant of the objects declared in the host language. Thus, if one wants to transfer a structure, or an array section, it will be necessary to provide in MPI a definition of a communication buffer that mimics the definition of the structure or array section in question. These facilities can be used by library designers to define communication functions that can transfer objects defined in the host language — by decoding their definitions as available in a symbol table or a dope vector. Such higher-level communication functions are not part of MPI.

More general communication buffers are specified by replacing the basic datatypes that have been used so far with derived datatypes that are constructed from basic datatypes using the constructors described in this section. These methods of constructing derived datatypes can be applied recursively.

1	A general datatype is an opaque object that specifies two things:
•	A sequence of basic datatypes
•	A sequence of integer (byte) displacements

7

9

15

16

17

18 19

2021

22

23

 24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

The displacements are not required to be positive, distinct, or in increasing order. Therefore, the order of items need not coincide with their order in store, and an item may appear more than once. We call such a pair of sequences (or sequence of pairs) a **type map**. The sequence of basic datatypes (displacements ignored) is the **type signature** of the datatype.

6 7

1

 $\mathbf{2}$

3

4

5

8 9

10

11 12

13

14

15

16

17

18

 $Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\$

be such a type map, where $type_i$ are basic types, and $disp_i$ are displacements. Let

```
Typesig = \{type_0, \dots, type_{n-1}\}
```

be the associated type signature. This type map, together with a base address **buf**, specifies a communication buffer: the communication buffer that consists of n entries, where the *i*-th entry is at address **buf** + $disp_i$ and has type $type_i$. A message assembled from such a communication buffer will consist of n values, of the types defined by Typesig.

Most datatype constructors have replication count or block length arguments. Allowed values are non-negative integers. If the value is zero, no elements are generated in the type map and there is no effect on datatype bounds or extent.

¹⁹ We can use a handle to a general datatype as an argument in a send or receive operation, ²⁰ instead of a basic datatype argument. The operation MPI_SEND(buf, 1, datatype,...) will ²² use the send buffer defined by the base address buf and the general datatype associated ²³ with datatype; it will generate a message with the type signature determined by the datatype ²⁴ argument. MPI_RECV(buf, 1, datatype,...) will use the receive buffer defined by the base ²⁵ address buf and the general datatype associated with datatype.

General datatypes can be used in all send and receive operations. We discuss, in Section 5.1.11, the case where the second argument count has value > 1.

The basic datatypes presented in Section 3.2.2 are particular cases of a general datatype, and are predefined. Thus, MPL_INT is a predefined handle to a datatype with type map $\{(\texttt{int}, 0)\}$, with one entry of type **int** and displacement zero. The other basic datatypes are similar.

The **extent** of a datatype is defined to be the span from the first byte to the last byte occupied by entries in this datatype, rounded up to satisfy alignment requirements. That is, if

$$Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\$$

then

32

33

34 35 36

42

$$lb(Typemap) = \min_{j} disp_{j},$$

$$ub(Typemap) = \max_{j} (disp_{j} + \text{sizeof}(type_{j})) + \epsilon, \text{ and}$$

$$extent(Typemap) = ub(Typemap) - lb(Typemap).$$
(5.1)

⁴³ If $type_j$ requires alignment to a byte address that is a multiple of k_j , then ϵ is the least ⁴⁴ non-negative increment needed to round extent(Typemap) to the next multiple of $\max_j k_j$. ⁴⁵ In Fortran, it is implementation dependent whether the MPI implementation computes ⁴⁶ the alignments k_j according to the alignments used by the compiler in common blocks, ⁴⁷ SEQUENCE derived types, BIND(C) derived types, or derived types that are neither SEQUENCE ⁴⁸ nor BIND(C). The complete definition of **extent** is given by Equation 5.1 Section 5.1.

Unofficial Draft for Comment Only

Let

Example 5.1 Assume that $Type = \{(double, 0), (char, 8)\}$ (a double at displacement zero, followed by a char at displacement eight). Assume, furthermore, that doubles have to be strictly aligned at addresses that are multiples of eight. Then, the extent of this datatype is 16 (9 rounded to the next multiple of 8). A datatype that consists of a character immediately followed by a double will also have an extent of 16.

Rationale. The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfill alignment constraints. More explicit control of the extent is provided in Section 5.1.6. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. In Fortran, structures can be expressed with several language features, e.g., common blocks, SEQUENCE derived types, or BIND(C) derived types. The compiler may use different alignments, and therefore, it is recommended to use MPI_TYPE_CREATE_RESIZED for arrays of structures if an alignment may cause an alignment-gap at the end of a structure as described in Section 5.1.6 and in Section 19.1.15. (End of rationale.)

5.1.1 Type Constructors with Explicit Addresses

In Fortran, the functions MPI_TYPE_CREATE_HVECTOR, MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_HINDEXED_BLOCK, MPI_TYPE_CREATE_STRUCT, and MPI_GET_ADDRESS accept arguments of type INTEGER(KIND=MPI_ADDRESS_KIND), wherever arguments of type MPI_Aint are used in C. For Fortran compilers that do not support the Fortran 90 KIND notation, and where addresses are 64 bits whereas default INTEGERs are 32 bits, these arguments will be of type INTEGER*8 (assuming the Fortran compiler accepts the common extension of INTEGER*8 for eight-byte integers).

5.1.2 Datatype Constructors

Contiguous The simplest datatype constructor is MPI_TYPE_CONTIGUOUS which allows replication of a datatype into contiguous locations.

		33
MPI_TYPE_CONTIGUOUS(count, oldtyp	e, newtype)	34
IN count	replication count (non-negative integer)	35
IN oldtype	old datatype (handle)	36
in oldspe		37
OUT newtype	new datatype (handle)	38
		39
C binding		40
int MPI_Type_contiguous(int count,	MPI_Datatype oldtype,	41
MPI_Datatype *newtype		42
		43
Fortran 2008 binding		44
MPI_Type_contiguous(count, oldtype	e, newtype, ierror)	45
INTEGER, INTENT(IN) :: count		46
TYPE(MPI_Datatype), INTENT(IN)	<i></i>	47
TYPE(MPI_Datatype), INTENT(OUT	C) :: newtype	48

1 2

3

4

5 6

 $\overline{7}$

8

9

10

11

12

13

14

15

16 17 18

19

20

21

22

23

24

25

26

27 28

29 30

31

1	INTE	GER, OPTIONAL, INTENT(OUT	C) :: ierror
2 3	Fortran	6	
4 5		_CONTIGUOUS(COUNT, OLDTYF GER COUNT, OLDTYPE, NEWTY	
6 7 8			by concatenating count copies of <i>extent</i> as the size of the concatenated copies.
9 10	-	5.2 Let oldtype have type n. The type map of the dataty	hap {(double, 0), (char, 8)}, with extent 16, and let pe returned by newtype is
11 12	{(dc	puble, 0), (char, 8), (double, 1)	$6), (\texttt{char}, 24), (\texttt{double}, 32), (\texttt{char}, 40)\};$
13 14	i.e., altern	nating double and char eleme	ents, with displacements $0, 8, 16, 24, 32, 40$.
14 15	In ge	neral, assume that the type m	nap of oldtype is
16 17	$\{(ty)$	$pe_0, disp_0), \ldots, (type_{n-1}, disp_n)$	$_{n-1})\},$
18 19	with exter	nt ex . Then newtype has a type	be map with $count \cdot n$ entries defined by:
19 20	$\{(type$	$(0, disp_0), \ldots, (type_{n-1}, disp_{n-1})$	$(type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex),$
21 22	$\ldots, (t_{\ell})$	$ype_0, disp_0 + ex \cdot (count - 1)),$	$\ldots, (type_{n-1}, disp_{n-1} + ex \cdot (count - 1))\}.$
23 24 25 26 27 28	cation of obtained	a datatype into locations the	TOR is a more general constructor that allows repli- at consist of equally spaced blocks. Each block is umber of copies of the old datatype. The spacing ent of the old datatype.
29 30	MPI_TYP	E_VECTOR(count, blocklengt	n, stride, oldtype, newtype)
31	IN	count	number of blocks (non-negative integer)
32 33	IN	blocklength	number of elements in each block (non-negative integer)
34 35 36	IN	stride	number of elements between start of each block (integer)
37	IN	oldtype	old datatype (handle)
38 39	OUT	newtype	new datatype (handle)
40	C bindin	g	
41 42	int MPI_	• •	t blocklength, int stride, e, MPI_Datatype *newtype)
43 44	Fortran	2008 binding	
45	MPI_Type	_vector(count, blocklengt	h, stride, oldtype, newtype, ierror)
46		GER, INTENT(IN) :: count,	-
47 48		<pre>(MPI_Datatype), INTENT(IN (MPI_Datatype), INTENT(OU)</pre>	

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
Fortran binding MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR	2 3 4 5 6
Example 5.3 Assume, again, that oldtype has type map {(double, 0), (char, 8)}, with extent 16. A call to MPI_TYPE_VECTOR(2, 3, 4, oldtype, newtype) will create the datatype with type map,	7 8 9
$\{(\texttt{double},0),(\texttt{char},8),(\texttt{double},16),(\texttt{char},24),(\texttt{double},32),(\texttt{char},40),$	10 11
$(\texttt{double}, 64), (\texttt{char}, 72), (\texttt{double}, 80), (\texttt{char}, 88), (\texttt{double}, 96), (\texttt{char}, 104) \}.$	12 13
That is, two blocks with three copies each of the old type, with a stride of 4 elements $(4 \cdot 16 \text{ bytes})$ between the the start of each block.	14 15 16
Example 5.4 A call to MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype) will create the datatype,	17 18 19
$\{(\texttt{double}, 0), (\texttt{char}, 8), (\texttt{double}, -32), (\texttt{char}, -24), (\texttt{double}, -64), (\texttt{char}, -56)\}.$	20
In general, assume that oldtype has type map,	21 22
$\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\$	23
with extent ex . Let bl be the blocklength. The newly created datatype has a type map with count \cdot bl \cdot n entries:	24 25 26
$\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1}),\$	27 28
$(type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots,$	29 30
$(type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),$	31 32
$(type_0, disp_0 + stride \cdot ex), \dots, (type_{n-1}, disp_{n-1} + stride \cdot ex), \dots,$	33
$(type_0, disp_0 + (stride + bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (stride + bl - 1) \cdot ex), \dots,$	34 35
$(type_0, disp_0 + stride \cdot (count - 1) \cdot ex), \dots,$	36 37
$(type_{n-1}, disp_{n-1} + stride \cdot (count - 1) \cdot ex), \dots,$	38 39
$(type_0, disp_0 + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots,$	40
	41 42
$(type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex)\}.$	43
A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to	44 45
MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1, count, n, oldtype, newtype), n arbitrary.	46
	47 48

```
1
      Hvector The function MPI_TYPE_CREATE_HVECTOR is identical to
\mathbf{2}
      MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements. The
3
      use for both types of vector constructors is illustrated in Section 5.1.14. (H stands for
4
      "heterogeneous").
5
6
      MPI_TYPE_CREATE_HVECTOR(count, blocklength, stride, oldtype, newtype)
7
8
        IN
                                                 number of blocks (non-negative integer)
                   count
9
        IN
                   blocklength
                                                  number of elements in each block (non-negative
10
                                                  integer)
11
        IN
                   stride
                                                  number of bytes between start of each block (integer)
12
13
        IN
                   oldtype
                                                 old datatype (handle)
14
        OUT
                                                  new datatype (handle)
                   newtype
15
16
      C binding
17
      int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,
18
                      MPI_Datatype oldtype, MPI_Datatype *newtype)
19
20
      Fortran 2008 binding
21
      MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
22
                       ierror)
23
           INTEGER, INTENT(IN) :: count, blocklength
24
           INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
25
           TYPE(MPI_Datatype), INTENT(IN) :: oldtype
26
           TYPE(MPI_Datatype), INTENT(OUT) :: newtype
27
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
      Fortran binding
29
      MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
30
                       IERROR)
31
           INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
32
           INTEGER(KIND=MPI_ADDRESS KIND) STRIDE
33
34
           Assume that oldtype has type map,
35
            \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\
36
37
      with extent ex. Let bl be the blocklength. The newly created datatype has a type map with
38
      \mathsf{count} \cdot \mathsf{bl} \cdot n entries:
39
40
            \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1}), 
41
42
            (type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots,
43
            (type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),
44
45
            (type_0, disp_0 + \mathsf{stride}), \ldots, (type_{n-1}, disp_{n-1} + \mathsf{stride}), \ldots,
46
47
            (type_0, disp_0 + stride + (bl - 1) \cdot ex), \ldots,
48
```

$(type_{n-1}, disp_{n-1} + stride + (bl - 1) \cdot ex), \ldots,$	
$(type_0, disp_0 + stride \cdot (count - 1)), \dots, (type_{n-1}, disp_{n-1} + stride \cdot (count - 1)), \dots,$;
$(type_0, disp_0 + stride \cdot (count - 1) + (bl - 1) \cdot ex), \dots,$	
$(type_{n-1}, disp_{n-1} + stride \cdot (count - 1) + (bl - 1) \cdot ex)\}.$	

Indexed The function MPI_TYPE_INDEXED allows replication of an old datatype into a sequence of blocks (each block is a concatenation of the old datatype), where each block can contain a different number of copies and have a different displacement. All block displacements are multiples of the old type extent.

MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype,

_	- (, , , , , , , , , , , , , , , , , ,	o , <i>y</i> ==1, , <i>y</i> , <i>y</i>	15
	newtype)		16
IN	count	number of blocks – also number of entries in	17
		$array_of_displacements and array_of_blocklengths$	18
		(non-negative integer)	19
IN	array_of_blocklengths	number of elements per block (array of non-negative	20
		integers)	21
IN	array_of_displacements	displacement for each block, in multiples of oldtype	22
	array_or_displacements	(array of integers)	23
		(array of integers)	24
IN	oldtype	old datatype (handle)	25
OUT	newtype	new datatype (handle)	26
			27
C binding	σ.		28
			29
int MPI_I		onst int array_of_blocklengths[],	30
		<pre>isplacements[], MPI_Datatype oldtype,</pre>	31
	MPI_Datatype *newtyp	e)	32
_			

Fortran 2008 binding

<pre>MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,</pre>	34
oldtype, newtype, ierror)	35
<pre>INTEGER, INTENT(IN) :: count, array_of_blocklengths(count),</pre>	36
array_of_displacements(count)	37
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	38
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	39
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
Fortron hinding	41
Fortran binding	42
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,	
	43

OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR

 44

1 2 3	(3, 1) and 1		p {(double, 0), (char, 8)}, with extent 16. Let $B = TYPE_INDEXED(2, B, D, oldtype, newtype)$ returns
4			$80), ({\tt char}, 88), ({\tt double}, 96), ({\tt char}, 104),$
5 6		ble, 0), (char, 8).	
7	× ×		ting at displacement 64, and one copy starting at
8 9	displaceme		
10 11	-	$(be_0, disp_0), \dots, (type_{n-1}, disp_n)$	
12 13			ocklengths argument and D be the ewly created datatype has $n \cdot \sum_{i=0}^{\text{count}-1} B[i]$ entries:
14 15		$pe_0, disp_0 + D[0] \cdot ex), \dots, (typ)$	
16	(type	$e_0, disp_0 + (D[0] + B[0] - 1) \cdot e_0$	x),,
17 18	(type	$e_{n-1}, disp_{n-1} + (D[0] + B[0] - $	$1) \cdot ex), \ldots,$
19	(type	$e_0, disp_0 + D[count-1] \cdot ex), \dots$	$(type_{n-1}, disp_{n-1} + D[count-1] \cdot ex), \ldots,$
20 21	(type	$x_0, disp_0 + (D[count-1] + B[count-1])$	$nt extsf{-1}] - 1) \cdot ex), \dots,$
22	(type	$a_{n-1}, disp_{n-1} + (D[count-1] + I)$	$B[count-1] - 1) \cdot ex)\}.$
23 24 25	A call to MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype) is equivalent to a call to MPI_TYPE_INDEXED(count, B, D, oldtype, newtype) where		
25 26	$D[j] = j \cdot stride, \ j = 0, \dots, count - 1,$		
27	and		
28 29	B[j] =	= blocklength, $j=0,\ldots,$ count	z = 1.
30 31	Hindexed	The function MPI_TYPE_CF	EATE_HINDEXED is identical to
32		E_INDEXED , except that block tes, rather than in multiples or	displacements in array_of_displacements are spec-
33 34	med m byt	tes, rather than in multiples of	the oldype extent.
35 36	MPI_TYPE	E_CREATE_HINDEXED(count, oldtype, newtype)	array_of_blocklengths, array_of_displacements,
37 38	IN	count	number of blocks – also number of entries in
39 40			array_of_displacements and array_of_blocklengths (non-negative integer)
41 42	IN	array_of_blocklengths	number of elements in each block (array of non-negative integers)
43	IN	array_of_displacements	byte displacement of each block (array of integers)
44 45	IN	oldtype	old datatype (handle)
46	OUT	newtype	new datatype (handle)
47 48	C binding	S	

1 int MPI_Type_create_hindexed(int count, const int array_of_blocklengths[], 2 const MPI_Aint array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype) 4 Fortran 2008 binding 5MPI_Type_create_hindexed(count, array_of_blocklengths, 6 array_of_displacements, oldtype, newtype, ierror) 7 INTEGER, INTENT(IN) :: count, array_of_blocklengths(count) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: 9 array_of_displacements(count) 10 TYPE(MPI_Datatype), INTENT(IN) :: oldtype 11 TYPE(MPI_Datatype), INTENT(OUT) :: newtype 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 14Fortran binding 15MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, 16ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) 17 INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR 18 INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*) 19 Assume that oldtype has type map, 2021 $\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},\$ 22 with extent ex. Let B be the array_of_blocklengths argument and D be the 23array_of_displacements argument. The newly created datatype has a type map with $n \cdot$ 24 $\sum_{i=0}^{\text{count}-1} B[i]$ entries: 2526 $\{(type_0, disp_0 + D[0]), \dots, (type_{n-1}, disp_{n-1} + D[0]), \dots, \}$ 2728 $(type_0, disp_0 + \mathsf{D}[\mathbf{0}] + (\mathsf{B}[\mathbf{0}] - 1) \cdot ex), \dots,$ 29 30 $(type_{n-1}, disp_{n-1} + D[0] + (B[0] - 1) \cdot ex), \dots,$ 31 $(type_0, disp_0 + \mathsf{D}[\mathsf{count-1}]), \dots, (type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count-1}]), \dots,$ 32 33 $(type_0, disp_0 + \mathsf{D}[\mathsf{count-1}] + (\mathsf{B}[\mathsf{count-1}] - 1) \cdot ex), \dots,$ 34 35 $(type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count-1}] + (\mathsf{B}[\mathsf{count-1}] - 1) \cdot ex)\}.$ 36 37 38

Indexed_block This function is the same as MPI_TYPE_INDEXED except that the blocklength is the same for all blocks. There are many codes using indirect addressing arising from unstructured grids where the blocksize is always 1 (gather/scatter). The following convenience function allows for constant blocksize and arbitrary displacements.

Unofficial Draft for Comment Only

39

40

41

```
MPI_TYPE_CREATE_INDEXED_BLOCK(count, blocklength, array_of_displacements,
              oldtype, newtype)
 IN
                                      length of array of displacements (non-negative
           count
                                      integer)
           blocklength
 IN
                                      size of block (non-negative integer)
 IN
           array_of_displacements
                                      array of displacements (array of integers)
 IN
           oldtype
                                      old datatype (handle)
 OUT
                                      new datatype (handle)
           newtype
C binding
int MPI_Type_create_indexed_block(int count, int blocklength,
               const int array_of_displacements[], MPI_Datatype oldtype,
              MPI_Datatype *newtype)
Fortran 2008 binding
MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
              oldtype, newtype, ierror)
    INTEGER, INTENT(IN) :: count, blocklength,
               array_of_displacements(count)
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
              OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
               NEWTYPE, IERROR
Hindexed_block The function MPI_TYPE_CREATE_HINDEXED_BLOCK is identical to
MPI_TYPE_CREATE_INDEXED_BLOCK, except that block displacements in
array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.
MPI_TYPE_CREATE_HINDEXED_BLOCK(count, blocklength, array_of_displacements,
              oldtype, newtype)
 IN
                                      length of array of displacements (non-negative
           count
                                      integer)
 IN
           blocklength
                                      size of block (non-negative integer)
 IN
           array_of_displacements
                                      byte displacement of each block (array of integers)
 IN
           oldtype
                                      old datatype (handle)
 OUT
           newtype
                                      new datatype (handle)
C binding
```

1

 $\mathbf{2}$

3

4

5

6 7

8

9

10 11

12

13

14 15

16

17

18

19

20

21

22

23

24 25

26

27

28

29 30 31

32

33

34 35 36

37

38

39

40 41

42

43

44 45

46 47

int MPI_7	<pre>Cype_create_hindexed_bloc</pre>	k(int count, int blocklength,	1
	·	_of_displacements[], MPI_Datatype oldtype,	2
	MPI_Datatype *newtyp	e)	3 4
Fortran 2	2008 binding		5
MPI_Type_		unt, blocklength, array_of_displacements,	6
	oldtype, newtype, ie		7
	ER, INTENT(IN) :: count,	5	8
INTEC	ER(KIND=MPI_ADDRESS_KIND) array_of_displacement		9
TYPE	(MPI_Datatype), INTENT(IN)		10
	(MPI_Datatype), INTENT(OU		11 12
INTEC	ER, OPTIONAL, INTENT(OUT)) :: ierror	13
Fortran b	binding		14
	6	UNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,	15
	OLDTYPE, NEWTYPE, IE		16
	ER COUNT, BLOCKLENGTH, O		17
INTEG	ER(KIND=MPI_ADDRESS_KIND)) ARRAY_OF_DISPLACEMENTS(*)	18 19
			20
Struct M	PI_TYPE_CREATE_STRUCT	is the most general type constructor. It further	21
generalizes	MPI_TYPE_CREATE_HINDE	XED in that it allows each block to consist of repli-	22
cations of	different datatypes.		23
			24
MPI_TYPI	E_CREATE_STRUCT(count, a	rray_of_blocklengths, array_of_displacements,	25
-	array_of_types, newtype)		26
IN	count	number of blocks also number of entries in arrays	27 28
		array_of_types, array_of_displacements, and	29
		array_of_blocklengths (non-negative integer)	30
IN	array_of_blocklengths	number of elements in each block (array of	31
		non-negative integers)	32
IN	array_of_displacements	byte displacement of each block (array of integers)	33 34
1N	array_of_types	types of elements in each block (array of handles)	34 35
Ουτ	newtype	new datatype (handle)	36
001	newtype	new datatype (nandle)	37
C binding	r		38
			39
_	const MPI_Aint array	• •	40
	const MPI_Datatype a	<pre>rray_of_types[], MPI_Datatype *newtype)</pre>	41
Fortran ?	2008 binding		42 43
	create_struct(count, arra	ay_of_blocklengths,	44
- 71 -		ts, array_of_types, newtype, ierror)	45
INTEC		array_of_blocklengths(count)	46
INTEC	ER(KIND=MPI_ADDRESS_KIND)		47
	array_of_displacement	nts(count)	48

Unofficial Draft for Comment Only

1 2 3	TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count) TYPE(MPI_Datatype), INTENT(OUT) :: newtype INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4 5 6 7 8 9	Fortran binding MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR) INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE, IERROR
10 11	INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
12	Example 5.6 Let type1 have type map,
$13 \\ 14$	$\{(\texttt{double}, 0), (\texttt{char}, 8)\},\$
15 16 17 18	with extent 16. Let $B = (2, 1, 3)$, $D = (0, 16, 26)$, and $T = (MPI_FLOAT, type1, MPI_CHAR)$. Then a call to MPI_TYPE_CREATE_STRUCT(3, B, D, T, newtype) returns a datatype with type map,
19	$\{(\texttt{float}, 0), (\texttt{float}, 4), (\texttt{double}, 16), (\texttt{char}, 24), (\texttt{char}, 26), (\texttt{char}, 27), (\texttt{char}, 28)\}.$
20 21 22 23	That is, two copies of MPI_FLOAT starting at 0, followed by one copy of type1 starting at 16, followed by three copies of MPI_CHAR, starting at 26. (We assume that a float occupies four bytes.) In general, let T be the array_of_types argument, where T[i] is a handle to,
24 25 26	$typemap_{i} = \{(type_{0}^{i}, disp_{0}^{i}), \dots, (type_{n_{i}-1}^{i}, disp_{n_{i}-1}^{i})\},\$
27 28 29	with extent ex_i . Let B be the array_of_blocklength argument and D be the array_of_displacements argument. Let c be the count argument. Then the newly created datatype has a type map with $\sum_{i=0}^{C-1} B[i] \cdot n_i$ entries:
30 31	$\{(type_0^0, disp_0^0 + D[0]), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0]), \dots, \}$
32 33	$(type_0^0, disp_0^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 $
34 35	$(type_0^{C-1}, disp_0^{C-1} + D[c-1]), \dots, (type_{n_{C-1}-1}^{C-1}, disp_{n_{C-1}-1}^{C-1} + D[c-1]), \dots,$
36 37	$(type_0^{\mathbf{C}-1}, disp_0^{\mathbf{C}-1} + \mathbf{D}[\mathbf{c}-1] + (\mathbf{B}[\mathbf{c}-1] - 1) \cdot ex_{\mathbf{C}-1}), \dots,$
38 39	$(type_{n_{C-1}-1}^{C-1}, disp_{n_{C-1}-1}^{C-1} + D[c-1] + (B[c-1]-1) \cdot ex_{C-1})\}.$
40 41 42 43	A call to MPI_TYPE_CREATE_HINDEXED(count, B, D, oldtype, newtype) is equivalent to a call to MPI_TYPE_CREATE_STRUCT(count, B, D, T, newtype), where each entry of T is equal to oldtype.
44	
$45 \\ 46$	
47	

 48

5.1. DERIVED DATATYPES

5.1.3 Subarray Datatype Constructor

MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

			6
IN	ndims	number of array dimensions (positive integer)	7
IN	array_of_sizes	number of elements of type $oldtype$ in each dimension	8
		of the full array (array of positive integers)	9
IN	array_of_subsizes	number of elements of type oldtype in each dimension	10
	-	of the subarray (array of positive integers)	11
IN	array_of_starts	starting coordinates of the subarray in each	12
		dimension (array of non-negative integers)	13
			14
IN	order	array storage order flag (state)	15
IN	oldtype	old datatype (handle)	16
OUT	newtype	new datatype (handle)	17
	51		18

C binding

<pre>int MPI_Type_create_subarray(int ndims, const int array_of_sizes[],</pre>
<pre>const int array_of_subsizes[], const int array_of_starts[],</pre>
int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

Fortran 2008 binding

<pre>MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,</pre>
array_of_starts, order, oldtype, newtype, ierror)
<pre>INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),</pre>
<pre>array_of_subsizes(ndims), array_of_starts(ndims), order</pre>
TYPE(MPI_Datatype), INTENT(IN) :: oldtype
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES, ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR) INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*), ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR

The subarray type constructor creates an MPI datatype describing an n-dimensional subarray of an n-dimensional array. The subarray may be situated anywhere within the full array, and may be of any nonzero size up to the size of the larger array as long as it is confined within this array. This type constructor facilitates creating filetypes to access arrays distributed in blocks among processes to a single file that contains the global array, see MPI I/O, especially Section 14.1.1.

This type constructor can handle arrays with an arbitrary number of dimensions and works for both C and Fortran ordered matrices (i.e., row-major or column-major). Note that a C program may use Fortran order and a Fortran program may use C order.

The ndims parameter specifies the number of dimensions in the full data array and gives the number of elements in array_of_sizes, array_of_subsizes, and array_of_starts.

Unofficial Draft for Comment Only

1 2 3 4	The number of elements of type oldtype in each dimension of the <i>n</i> -dimensional array and the requested subarray are specified by array_of_sizes and array_of_subsizes, respectively. For any dimension i, it is erroneous to specify array_of_subsizes[i] < 1 or array_of_subsizes[i] > array_of_sizes[i].
5 6 7 8	The array_of_starts contains the starting coordinates of each dimension of the subarray. Arrays are assumed to be indexed starting from zero. For any dimension i , it is erroneous to specify array_of_starts[i] < 0 or array_of_starts[i] > (array_of_sizes[i] - array_of_subsizes[i]).
9 10 11 12	Advice to users. In a Fortran program with arrays indexed starting from 1, if the starting coordinate of a particular dimension of the subarray is n, then the entry in array_of_starts for that dimension is n-1. (End of advice to users.)
12 13 14	The order argument specifies the storage order for the subarray as well as the full array. It must be set to one of the following:
15 16	MPI_ORDER_C The ordering used by C arrays, (i.e., row-major order)
17	MPI_ORDER_FORTRAN The ordering used by Fortran arrays, (i.e., column-major order)
18 19 20	A ndims-dimensional subarray (newtype) with no extra padding can be defined by the function Subarray() as follows:
21 22 23 24	$\begin{array}{llllllllllllllllllllllllllllllllllll$
25 26	Let the typemap of oldtype have the form:
27 28	$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}$
29 30 31 32 33 34	where $type_i$ is a predefined MPI datatype, and let ex be the extent of oldtype. Then we define the Subarray() function recursively using the following three equations. Equation 5.2 defines the base step. Equation 5.3 defines the recursion step when order = MPI_ORDER_FORTRAN, and Equation 5.4 defines the recursion step when order = MPI_ORDER_C. These equations use the conceptual datatypes lb_marker and ub_marker; see Section 5.1.6 for details.
35 36	$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, $ (5.2)
37	$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\})$
38	$= \{(lb_marker, 0),$
39 40	$(type_0, disp_0 + start_0 \times ex), \dots, (type_{n-1}, disp_{n-1} + start_0 \times ex),$
40	$(type_0, disp_0 + (start_0 + 1) \times ex), \dots, (type_{n-1},$
42	$disp_{n-1} + (start_0 + 1) \times ex), \dots$
43	$(type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \ldots,$
44	$(type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex),$
45 46	$(ub_marker, size_0 \times ex)$ }
40	
48	Subarray($ndims$, { $size_0, size_1, \dots, size_{ndims-1}$ }, (5.3)

Unofficial Draft for Comment Only

$ \{start_0, start_1, \dots, start_{ndims-1}\}, oldtype) $ $ = Subarray(ndims - 1, \{size_1, size_2, \dots, size_{ndims-1}\}, \{subsize_1, subsize_2, \dots, subsize_{ndims-1}\}, \{start_1, start_2, \dots, start_{ndims-1}\}, \{start_1, start_2, \dots, start_{ndims-1}\}, \{start_0\}, oldtype)) $ $ Subarray(ndims, \{size_0, size_1, \dots, size_{ndims-1}\}, \{subsize_0, subsize_1, \dots, subsize_{ndims-1}\}, \{start_0, start_1, \dots, start_{ndims-1}\}, oldtype) $ $ = Subarray(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, \{start_0, start_1, \dots, start_{ndims-2}\}, \{subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype)) $		1
$ \{start_0, start_1, \dots, start_{ndims-1}\}, oldtype) $ $ = Subarray(ndims - 1, \{size_1, size_2, \dots, size_{ndims-1}\}, \{subsize_1, subsize_2, \dots, subsize_{ndims-1}\}, \{start_1, start_2, \dots, start_{ndims-1}\}, \{start_1, start_2, \dots, start_{ndims-1}\}, \{start_0\}, oldtype)) $ $ Subarray(ndims, \{size_0, size_1, \dots, size_{ndims-1}\}, \{subsize_0, subsize_1, \dots, subsize_{ndims-1}\}, \{start_0, start_1, \dots, start_{ndims-1}\}, oldtype) $ $ = Subarray(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, \{start_0, start_1, \dots, start_{ndims-2}\}, \{subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype)) $	$\{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\$	1
$= Subarray(ndims - 1, \{size_1, size_2, \dots, size_{ndims-1}\}, \{subsize_1, subsize_2, \dots, subsize_{ndims-1}\}, \{start_1, start_2, \dots, start_{ndims-1}\}, \{start_1, start_2, \dots, start_{ndims-1}\}, \{subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype))$ $Subarray(ndims, \{size_0, size_1, \dots, size_{ndims-1}\}, \{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$ $= Subarray(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, \{subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype))$	$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$	2
$ \{subsize_1, subsize_2, \dots, subsize_{ndims-1}\}, \\ \{start_1, start_2, \dots, start_{ndims-1}\}, \\ Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype)) \end{cases} $ $ Subarray(ndims, \{size_0, size_1, \dots, size_{ndims-1}\}, \\ \{subsize_0, subsize_1, \dots, subsize_{ndims-1}\}, \\ \{start_0, start_1, \dots, start_{ndims-1}\}, oldtype) $ $ = Subarray(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, \\ \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, \\ \{subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype)) $	= Subarray($ndims - 1, \{size_1, size_2, \dots, size_{ndims-1}\},\$	3
$ \{start_1, start_2, \dots, start_{ndims-1}\}, $ $ Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype)) $ $ Subarray(ndims, \{size_0, size_1, \dots, size_{ndims-1}\}, $ $ \{subsize_0, subsize_1, \dots, subsize_{ndims-1}\}, $ $ \{start_0, start_1, \dots, start_{ndims-1}\}, oldtype) $ $ = Subarray(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, $ $ \{subsize_0, subsize_1, \dots, start_{ndims-2}\}, $ $ \{start_0, start_1, \dots, start_{ndims-2}\}, $ $ \{subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype)) $		4
$\begin{aligned} & \text{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype)) \end{aligned} \\ & \text{Subarray}(ndims, \{size_0, size_1, \dots, size_{ndims-1}\}, \\ & \{subsize_0, subsize_1, \dots, subsize_{ndims-1}\}, \\ & \{start_0, start_1, \dots, start_{ndims-1}\}, oldtype) \end{aligned} \\ & = & \text{Subarray}(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, \\ & \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, \\ & \{start_0, start_1, \dots, start_{ndims-2}\}, \\ & \{subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype)) \end{aligned}$	$\{suosize_1, suosize_2, \dots, suosize_{ndims-1}\},\$	5
$\begin{aligned} & \text{Subarray}(ndims, \{size_0, size_1, \dots, size_{ndims-1}\}, \\ & \{subsize_0, subsize_1, \dots, subsize_{ndims-1}\}, \\ & \{start_0, start_1, \dots, start_{ndims-1}\}, \text{oldtype}) \end{aligned} $ $= & \text{Subarray}(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, \\ & \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, \\ & \{start_0, start_1, \dots, start_{ndims-2}\}, \\ & \{start_0, start_1, \dots, start_{ndims-2}\}, \\ & \text{Subarray}(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, \text{oldtype})) \end{aligned}$	$\{start_1, start_2, \dots, start_{ndims-1}\},\$	6
$\begin{aligned} & \text{Subarray}(ndims, \{size_0, size_1, \dots, size_{ndims-1}\}, \\ & \{subsize_0, subsize_1, \dots, subsize_{ndims-1}\}, \\ & \{start_0, start_1, \dots, start_{ndims-1}\}, \text{oldtype}) \end{aligned} $ $= & \text{Subarray}(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, \\ & \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, \\ & \{start_0, start_1, \dots, start_{ndims-2}\}, \\ & \{start_0, start_1, \dots, start_{ndims-2}\}, \\ & \text{Subarray}(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, \text{oldtype})) \end{aligned}$	$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype))$	7
$ \begin{aligned} & \text{Subarray}(ndims, \{size_0, size_1, \dots, size_{ndims-1}\}, & (5.4) \\ & \{subsize_0, subsize_1, \dots, subsize_{ndims-1}\}, & (5.4) \\ & \{start_0, start_1, \dots, start_{ndims-1}\}, \text{oldtype}) \\ & = & \text{Subarray}(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, & (5.4) \\ & \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, & (5.4) \\ & \{start_0, start_1, \dots, start_{ndims-2}\}, & (5.4) \\ & \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, & (5.4) \\ & \{subsize_0, subsize_1, \dots, size_{ndims-2}\}, & (5.4) \\ & \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, & (5.4) \\ & \{subsize_0, subsize_{ndims-1}\}, & (5.4) \\ & \{subsize_0, subsize_{ndims-1}\}, & (5.4) \\ & \{subsize_{ndims-1}\}, & (5.4) \\ & (5.4) \\ & (5.4) \\ & (5.4) \\ & (5.4) \\ & (5.4) \\$		8
$ \{subsize_0, subsize_1, \dots, subsize_{ndims-1}\}, \\ \{start_0, start_1, \dots, start_{ndims-1}\}, oldtype) \\ = Subarray(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, \\ \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, \\ \{start_0, start_1, \dots, start_{ndims-2}\}, \\ \{subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype)) \\ \end{bmatrix} $		9
$ \{start_0, start_1, \dots, start_{ndims-1}\}, oldtype) $ $ = Subarray(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, $ $ \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, $ $ \{start_0, start_1, \dots, start_{ndims-2}\}, $ $ Subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype)) $	Subarray($ndims$, { $size_0, size_1, \dots, size_{ndims-1}$ }, (5.4)	10
$= Subarray(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, \{start_0, start_1, \dots, start_{ndims-2}\}, \\ \{subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype)) $	$\{subsize_0, subsize_1, \dots, subsize_{ndims-1}\},\$	11
$= Subarray(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, \{start_0, start_1, \dots, start_{ndims-2}\}, \{start_0, start_1, \dots, start_{ndims-2}\}, Subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype))$	$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$	12
$ \{subsize_0, subsize_1, \dots, subsize_{ndims-2}\}, \\ \{start_0, start_1, \dots, start_{ndims-2}\}, \\ Subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype)) $	= Subarray($ndims - 1$, { $size_0, size_1, \dots, size_{ndime_2}$ }.	13
$\{start_0, start_1, \dots, start_{ndims-2}\},$ Subarray(1, { $size_{ndims-1}$ }, { $subsize_{ndims-1}$ }, { $start_{ndims-1}$ }, oldtype))		14
$Subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype))$	$\{subsize_0, subsize_1, \dots, subsize_{ndims-2}\},\$	15
	$\{start_0, start_1, \dots, start_{ndims-2}\},\$	16
1	$Subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype))$	17
		18

For an example use of MPI_TYPE_CREATE_SUBARRAY in the context of I/O see Section 14.9.2.

5.1.4 Distributed Array Datatype Constructor

The distributed array type constructor supports HPF-like [47] data distributions. However, unlike in HPF, the storage order may be specified for C arrays as well as for Fortran arrays.

Advice to users. One can create an HPF-like file view using this type constructor as follows. Complementary filetypes are created by having every process of a group call this constructor with identical arguments (with the exception of rank which should be set appropriately). These filetypes (along with identical disp and etype) are then used to define the view (via MPI_FILE_SET_VIEW), see MPI I/O, especially Section 14.1.1 and Section 14.3. Using this view, a collective data access operation (with identical offsets) will yield an HPF-like distribution pattern. (*End of advice to users.*)

 24

 31

	124		CHAI TER 9. DATAT ITES		
1 2	MPI_TYPE		k, ndims, array_of_gsizes, array_of_distribs, _psizes, order, oldtype, newtype)		
3	IN	size	size of process group (positive integer)		
4 5	IN	rank	rank in process group (non-negative integer)		
6 7	IN	ndims	number of array dimensions as well as process grid dimensions (positive integer)		
8 9 10	IN	array_of_gsizes	number of elements of type oldtype in each dimension of global array (array of positive integers)		
11 12	IN	array_of_distribs	distribution of array in each dimension (array of states)		
13 14	IN	array_of_dargs	distribution argument in each dimension (array of positive integers)		
15 16	IN	array_of_psizes	size of process grid in each dimension (array of positive integers)		
17 18	IN	order	array storage order flag (state)		
19	IN	oldtype	old datatype (handle)		
20 21	OUT	newtype	new datatype (handle)		
23 24 25 26 27	<pre>int MPI_Type_create_darray(int size, int rank, int ndims,</pre>				
28	Fortran 2	008 binding			
29	MPI_Type_		, ndims, array_of_gsizes,		
30	array_of_distribs, array_of_dargs, array_of_psizes, order,				
31 32	TNTEG	oldtype, newtype, ie ER INTENT(IN) ·· size	rror) rank, ndims, array_of_gsizes(ndims),		
33	111110		dims), array_of_dargs(ndims),		
34		array_of_psizes(ndi			
35 36		MPI_Datatype), INTENT(IN			
37	TYPE(MPI_Datatype), INTENT(OUT) :: newtype INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
38					
39	Fortran binding MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,				
40 41	ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER,				
42	OLDTYPE, NEWTYPE, IERROR)				
43	<pre>INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),</pre>				
44		NEWTYPE, IERROR			
45 46	MPI T	TYPE CREATE DARRAY car	be used to generate the datatypes corresponding		
47	to the distribution of an ndims-dimensional array of oldtype elements onto an				
48	ndims-dimensional grid of logical processes. Unused dimensions of array_of_psizes should be				

set to 1. (See Example 5.7.) For a call to MPI_TYPE_CREATE_DARRAY to be correct, the equation $\prod_{i=0}^{ndims-1} array_of_psizes[i] = size$ must be satisfied. The ordering of processes in the process grid is assumed to be row-major, as in the case of virtual Cartesian process topologies.

Advice to users. For both Fortran and C arrays, the ordering of processes in the process grid is assumed to be row-major. This is consistent with the ordering used in virtual Cartesian process topologies in MPI. To create such virtual process topologies, or to find the coordinates of a process in the process grid, etc., users may use the corresponding process topology functions, see Chapter 8. (*End of advice to users.*)

Each dimension of the array can be distributed in one of three ways:

- MPI_DISTRIBUTE_BLOCK Block distribution
- MPI_DISTRIBUTE_CYCLIC Cyclic distribution
- MPI_DISTRIBUTE_NONE Dimension not distributed.

The constant MPI_DISTRIBUTE_DFLT_DARG specifies a default distribution argument. The distribution argument for a dimension that is not distributed is ignored. For any dimension i in which the distribution is MPI_DISTRIBUTE_BLOCK, it is erroneous to specify array_of_dargs[i] * array_of_psizes[i] < array_of_gsizes[i].

For example, the HPF layout ARRAY(CYCLIC(15)) corresponds to MPI_DISTRIBUTE_CYCLIC with a distribution argument of 15, and the HPF layout AR-RAY(BLOCK) corresponds to MPI_DISTRIBUTE_BLOCK with a distribution argument of MPI_DISTRIBUTE_DFLT_DARG.

The order argument is used as in MPI_TYPE_CREATE_SUBARRAY to specify the storage order. Therefore, arrays described by this type constructor may be stored in Fortran (column-major) or C (row-major) order. Valid values for order are MPI_ORDER_FORTRAN and MPI_ORDER_C.

This routine creates a new MPI datatype with a typemap defined in terms of a function called "cyclic()" (see below).

Without loss of generality, it suffices to define the typemap for the MPI_DISTRIBUTE_CYCLIC case where MPI_DISTRIBUTE_DFLT_DARG is not used.

MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_NONE can be reduced to the MPI_DISTRIBUTE_CYCLIC case for dimension i as follows.

MPI_DISTRIBUTE_BLOCK with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to

```
(array_of_gsizes[i] + array_of_psizes[i] - 1)/array_of_psizes[i].
```

If array_of_dargs[i] is not MPI_DISTRIBUTE_DFLT_DARG, then MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_CYCLIC are equivalent.

MPI_DISTRIBUTE_NONE is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to array_of_gsizes[i].

Finally, MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] equal to44MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with45array_of_dargs[i] set to 1.46

For MPI_ORDER_FORTRAN, an ndims-dimensional distributed array (newtype) is defined ⁴⁷ by the following code fragment: ⁴⁸

1

2

3 4

5

6

7

8

9

10 11

12

13 14

15 16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

31

32

33

34

35

36

37 38

39

40 41

42

```
1
          oldtypes[0] = oldtype;
\mathbf{2}
          for (i = 0; i < ndims; i++) {</pre>
3
               oldtypes[i+1] = cyclic(array_of_dargs[i],
4
                                            array_of_gsizes[i],
5
                                            r[i],
6
                                            array_of_psizes[i],
7
                                            oldtypes[i]);
8
          }
9
          newtype = oldtypes[ndims];
10
11
          For MPI_ORDER_C, the code is:
12
          oldtypes[0] = oldtype;
13
          for (i = 0; i < ndims; i++) {</pre>
14
               oldtypes[i + 1] = cyclic(array_of_dargs[ndims - i - 1],
15
                                              array_of_gsizes[ndims - i - 1],
16
                                              r[ndims - i - 1],
17
                                              array_of_psizes[ndims - i - 1],
18
                                              oldtypes[i]);
19
          }
20
          newtype = oldtypes[ndims];
21
22
23
     where r[i] is the position of the process (with rank rank) in the process grid at dimension i.
24
      The values of r[i] are given by the following code fragment:
25
26
          t_rank = rank;
27
          t_size = 1;
28
          for (i = 0; i < ndims; i++)
29
               t_size *= array_of_psizes[i];
30
          for (i = 0; i < ndims; i++) {</pre>
31
               t_size = t_size / array_of_psizes[i];
32
               r[i] = t_rank / t_size;
33
               t_rank = t_rank % t_size;
34
35
36
      Let the typemap of oldtype have the form:
37
           \{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}
38
39
      where type_i is a predefined MPI datatype, and let ex be the extent of
40
      oldtype. The following function uses the conceptual datatypes lb_marker and ub_marker, see
41
      Section 5.1.6 for details.
42
          Given the above, the function cyclic() is defined as follows:
43
44
           cyclic(darg, gsize, r, psize, oldtype)
45
             = {(lb_marker, 0),
46
                 (type_0, disp_0 + r \times darg \times ex), \ldots,
47
48
                         (type_{n-1}, disp_{n-1} + r \times darg \times ex),
```

$(type_0, disp_0 + (r \times darg + 1) \times ex), \dots,$	1
$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex),$	2
	3
$(type_0, disp_0 + ((r+1) \times darq - 1) \times ex), \ldots,$	4 5
$(type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex),$	6
$(vgpc_{n-1}, uvsp_{n-1} + ((r+1) \land uurg = 1) \land cv),$	7
	8
$(type_0, disp_0 + r imes darg imes ex + psize imes darg imes ex), \dots,$	9
$(type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex),$	10
$(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex), \dots,$	11
$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex),$	12
	13
$(type_0, disp_0 + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \dots,$	14 15
$(type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex),$	16
	17
$(t_{1}, \ldots, t_{n}) \in (t_{1}, \ldots, t_{n}) \in (t_{1}, \ldots, t_{n}) \in (t_{1}, \ldots, t_{n}) \in (t_{1}, \ldots, t_{n})$	18
$(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \dots,$	19
$(type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)),$	20
$(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex \times (count - 1)), \dots,$	21
$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex$	22 23
+psize imes darg imes ex imes (count-1)),	23
	25
$(type_0, disp_0 + (r \times darg + darg_{last} - 1) \times ex$	26
$+psize \times darg \times ex \times (count - 1)), \dots,$	27
$(type_{n-1}, disp_{n-1} + (r \times darg + darg_{last} - 1) \times ex$	28
+psize imes darg imes ex imes (count - 1)),	29
$(ub_marker, gsize * ex)$	30 31
	32
where <i>count</i> is defined by this code fragment:	33
<pre>nblocks = (gsize + (darg - 1)) / darg;</pre>	34
<pre>count = nblocks / psize;</pre>	35
<pre>left_over = nblocks - count * psize;</pre>	36
if (r < left_over)	37
<pre>count = count + 1;</pre>	38
Here, <i>nblocks</i> is the number of blocks that must be distributed among the processors.	39 40
Finally, $darg_{last}$ is defined by this code fragment:	40
if ((num_in_last_cyclic = gsize % (psize * darg)) == 0)	42

```
if ((num_in_last_cyc
                          = gsize % (psize * darg))
                                                       = 0)
    darg_last = darg;
else {
    darg_last = num_in_last_cyclic - darg * r;
    if (darg_last > darg)
        darg_last = darg;
    if (darg_last <= 0)</pre>
```

Unofficial Draft for Comment Only

43

44

45

46

47

```
1
                   darg_last = darg;
\mathbf{2}
              }
3
4
     Example 5.7 Consider generating the filetypes corresponding to the HPF distribution:
5
6
            <oldtype> FILEARRAY(100, 200, 300)
7
     !HPF$ PROCESSORS PROCESSES(2, 3)
8
     !HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
9
     This can be achieved by the following Fortran code, assuming there will be six processes
10
     attached to the run:
11
12
     ndims = 3
13
     array_of_gsizes(1) = 100
14
     array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
15
     array_of_dargs(1) = 10
16
     array_of_gsizes(2) = 200
17
     array_of_distribs(2) = MPI_DISTRIBUTE_NONE
18
     \operatorname{array_of_dargs}(2) = 0
19
     array_of_gsizes(3) = 300
20
     array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
21
     array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_DARG
22
     array_of_psizes(1) = 2
23
     array_of_psizes(2) = 1
24
     array_of_psizes(3) = 3
25
     call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
26
     call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
27
     call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
28
           array_of_distribs, array_of_dargs, array_of_psizes,
                                                                             &
29
           MPI_ORDER_FORTRAN, oldtype, newtype, ierr)
30
^{31}
            Address and Size Functions
     5.1.5
32
33
```

The displacements in a general datatype are relative to some initial buffer address. Absolute addresses can be substituted for these displacements: we treat them as displacements relative to "address zero," the start of the address space. This initial address zero is indicated by the constant MPI_BOTTOM. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the buf argument is passed the value MPI_BOTTOM. Note that in Fortran MPI_BOTTOM is not usable for initialization or assignment, see Section 2.5.4.

The address of a location in memory can be found by invoking the function MPI_GET_ADDRESS. The relative displacement between two absolute addresses can be calculated with the function MPI_AINT_DIFF. A new absolute address as sum of an absolute base address and a relative displacement can be calculated with the function MPI_AINT_ADD. To ensure portability, arithmetic on absolute addresses should not be

⁴⁵ performed with the intrinsic operators "-" and "+". See also Sections 2.5.6 and 5.1.12 on ⁴⁶ pages 20 and 143.

Address sized integer values, i.e., MPI_Aint or 1 Rationale. $\mathbf{2}$ INTEGER(KIND=MPI_ADDRESS_KIND) values, are signed integers, while absolute ad-3 dresses are unsigned quantities. Direct arithmetic on addresses stored in address sized signed variables can cause overflows, resulting in undefined behavior. (End of 4 rationale.) 56 7 MPI_GET_ADDRESS(location, address) 9 IN location location in caller memory (choice) 10 11 OUT address address of location (integer) 1213 C binding 14int MPI_Get_address(const void *location, MPI_Aint *address) 15Fortran 2008 binding 16MPI_Get_address(location, address, ierror) 17 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: location 18 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address 19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2021Fortran binding 22 MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR) 23<type> LOCATION(*) 24INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS 25INTEGER IERROR 26Returns the (byte) address of location. 2728 Rationale. In the mpi_f08 module, the location argument is not defined with 29 INTENT(IN) because existing applications may use MPI_GET_ADDRESS as a substi-30 tute for MPI_F_SYNC_REG, which was not defined before MPI-3.0. (End of rationale.) 3132 33 **Example 5.8** Using MPI_GET_ADDRESS for an array. 34 35REAL A(100,100) 36 INTEGER(KIND=MPI_ADDRESS_KIND) I1, I2, DIFF 37 CALL MPI_GET_ADDRESS(A(1,1), I1, IERROR) 38 CALL MPI_GET_ADDRESS(A(10,10), I2, IERROR) 39 DIFF = MPI_AINT_DIFF(I2, I1) 40 ! The value of DIFF is 909*SIZEOF(REAL); the values of I1 and I2 are 41 ! implementation dependent. 42Advice to users. C users may be tempted to avoid the usage of 43 MPI_GET_ADDRESS and rely on the availability of the address operator &. Note,

MPI_GET_ADDRESS and rely on the availability of the address operator &. Note, however, that & cast-expression is a pointer, not an address. ISO C does not require that the value of a pointer (or the pointer cast to int) be the absolute address of the object pointed at — although this is commonly the case. Furthermore, referencing may not have a unique definition on machines with a segmented address space. The

Unofficial Draft for Comment Only

		RESS to "reference" C variables guarantees portability to such
machines as well. (End of advice to users.)		
op		to prevent problems with the argument copying and register Fortran compilers, please note the hints in Sections $19.1.10-e\ to\ users.)$
	1 0,	arithmetic on MPI addresses must be performed using the AINT_DIFF functions.
MPI_AI	NT_ADD(base, disp)	
IN	base	base address (integer)
IN	disp	displacement (integer)
C bind		PI_Aint base, MPI_Aint disp)
	n 2008 binding	S_KIND) MPI_Aint_add(base, disp)
		DRESS_KIND), INTENT(IN) :: base, disp
Fortrai	n binding	
	•	S_KIND) MPI_AINT_ADD(BASE, DISP)
		DRESS_KIND) BASE, DISP
the base MPI_GE dress is in the s formed	e and disp argument ET_ADDRESS and divide a constraint of the pro- ame object references in a manner that resu	aces a new MPI_Aint value that is equivalent to the sum of s, where base represents a base address returned by a call to sp represents a signed integer displacement. The resulting ad- base that generated base, and it must correspond to a location bed by base, as described in Section 5.1.12. The addition is per- ilts in the correct MPI_Aint representation of the output address, ally produced base had called:
MPI_Get	t_address((char *)) base + disp, &result);
		•
	NT_DIFF(addr1, add	,
IN	addr1	minuend address (integer)
IN	addr2	subtrahend address (integer)
C bind 1PI_Air	0	MPI_Aint addr1, MPI_Aint addr2)
INTEGE		S_KIND) MPI_Aint_diff(addr1, addr2) DRESS_KIND), INTENT(IN) :: addr1, addr2
	n binding	
	3	

INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_DIFF(ADDR1, ADDR2) INTEGER(KIND=MPI_ADDRESS_KIND) ADDR1, ADDR2

MPI_AINT_DIFF produces a new MPI_Aint value that is equivalent to the difference between addr1 and addr2 arguments, where addr1 and addr2 represent addresses returned by calls to MPI_GET_ADDRESS. The resulting address is valid only at the process that generated addr1 and addr2, and addr1 and addr2 must correspond to locations in the same object in the same process, as described in Section 5.1.12. The difference is calculated in a manner that results in the signed difference from addr1 to addr2, as if the process that originally produced the addresses had called (char *) addr1 - (char *) addr2 on the addresses initially passed to MPI_GET_ADDRESS.

The following auxiliary functions provide useful information on derived datatypes.

MPI_TYPE_SIZE(datatype, size) IN datatype datatype (handle) OUT datatype size (integer) size C binding int MPI_Type_size(MPI_Datatype datatype, int *size) Fortran 2008 binding MPI_Type_size(datatype, size, ierror) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR) INTEGER DATATYPE, SIZE, IERROR MPI_TYPE_SIZE_X(datatype, size) IN datatype datatype (handle) OUT size datatype size (integer) C binding int MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size) Fortran 2008 binding MPI_Type_size_x(datatype, size, ierror) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR) INTEGER DATATYPE, IERROR

1

 $\mathbf{2}$

3

4

5

6

7

8

9

10

11

12 13 14

15

16

17 18 19

20

21

22

23

24

25

26 27

28

29

30 31 32

33

34 35

36 37

38

39

40

41

42

43

44 45

46

47

1 $\mathbf{2}$

3

4

5

6

7

8 9

10

INTEGER(KIND=MPI_COUNT_KIND) SIZE

MPI_TYPE_SIZE and MPI_TYPE_SIZE_X set the value of size to the total size, in bytes, of the entries in the type signature associated with datatype; i.e., the total size of the data in a message that would be created with this datatype. Entries that occur multiple times in the datatype are counted with their multiplicity. For both functions, if the OUT parameter cannot express the value to be returned (e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED.

Lower-Bound and Upper-Bound Markers 5.1.6

11It is often convenient to define explicitly the lower bound and upper bound of a type map, 12and override the definition given on page 132. This allows one to define a datatype that has 13 "holes" at its beginning or its end, or a datatype with entries that extend above the upper 14bound or below the lower bound. Examples of such usage are provided in Section 5.1.14. 15Also, the user may want to overide the alignment rules that are used to compute upper 16bounds and extents. E.g., a C compiler may allow the user to overide default alignment 17rules for some of the structures within a program. The user has to specify explicitly the 18 bounds of the datatypes that match these structures.

19To achieve this, we add two additional conceptual datatypes, **lb_marker** and 20**ub_marker**, that represent the lower bound and upper bound of a datatype. These con-21ceptual datatypes occupy no space $(extent(lb_marker) = extent(ub_marker) = 0)$. They do 22not affect the size or count of a datatype, and do not affect the content of a message created 23with this datatype. However, they do affect the definition of the extent of a datatype and, 24 therefore, affect the outcome of a replication of this datatype by a datatype constructor. 25

26**Example 5.9** A call to MPI_TYPE_CREATE_RESIZED(MPI_INT, -3, 9, type1) creates a new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer 2728 at displacement 0. This is the datatype defined by the typemap $\{(\mathsf{lb}_\mathsf{marker}, -3), (int, 0), \}$ 29(ub_marker, 6)}. If this type is replicated twice by a call to MPI_TYPE_CONTIGUOUS(2, 30 type1, type2) then the newly created type can be described by the typemap {(lb_marker, 31 -3), (int, 0), (int,9), (ub_marker, 15). (An entry of type ub_marker can be deleted if there 32 is another entry of type ub_marker with a higher displacement; an entry of type lb_marker 33can be deleted if there is another entry of type lb_marker with a lower displacement.) 34

In general, if

$$Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\$$

then the **lower bound** of Typemap is defined to be

$$lb(Typemap) = \begin{cases} \min_{j} disp_{j} & \text{if no entry has type} \\ \min_{j} \{ disp_{j} \text{ such that } type_{j} = |b_marker \} & \text{otherwise} \end{cases}$$

Similarly, the **upper bound** of *Typemap* is defined to be

$$ub(Typemap) = \begin{cases} \max_{j}(disp_{j} + sizeof(type_{j})) + \epsilon & \text{if no entry has type} \\ \max_{j}\{disp_{j} \text{ such that } type_{j} = \mathsf{ub_marker} \} & \text{otherwise} \end{cases}$$

46Then 47

35 36 37

38

39 40 41

42

43 4445

48

extent(Typemap) = ub(Typemap) - lb(Typemap)

Unofficial Draft for Comment Only

If $type_i$ requires alignment to a byte address that is a multiple of k_i , then ϵ is the least non-negative increment needed to round extent(Typemap) to the next multiple of $\max_i k_i$. In Fortran, it is implementation dependent whether the MPI implementation computes the alignments k_i according to the alignments used by the compiler in common blocks, SEQUENCE derived types, BIND(C) derived types, or derived types that are neither SEQUENCE nor BIND(C).

The formal definitions given for the various datatype constructors apply now, with the amended definition of **extent**.

Rationale. Before Fortran 2003, MPI_TYPE_CREATE_STRUCT could be applied to Fortran common blocks and SEQUENCE derived types. With Fortran 2003, this list was extended by BIND(C) derived types and MPI implementors have implemented the alignments k_i differently, i.e., some based on the alignments used in SEQUENCE derived types, and others according to BIND(C) derived types. (End of rationale.)

Advice to implementors. In Fortran, it is generally recommended to use BIND(C) derived types instead of common blocks or SEQUENCE derived types. Therefore it is recommended to calculate the alignments k_i based on BIND(C) derived types. (End of advice to implementors.)

Advice to users. Structures combining different basic datatypes should be defined so that there will be no gaps based on alignment rules. If such a datatype is used to create an array of structures, users should also avoid an alignment-gap at the end of the structure. In MPI communication, the content of such gaps would not be communicated into the receiver's buffer. For example, such an alignment-gap may occur between an odd number of floats or REALs before a double or DOUBLE PRECISION data. Such gaps may be added explicitly to both the structure and the MPI derived datatype handle because the communication of a contiguous derived datatype may be significantly faster than the communication of one that is non-contiguous because of such alignment-gaps.

Example: Instead of

```
TYPE, BIND(C) :: my_data
  REAL, DIMENSION(3) :: x
  ! there may be a gap of the size of one REAL
  ! if the alignment of a DOUBLE PRECISION is
  ! two times the size of a REAL
  DOUBLE PRECISION :: p
END TYPE
one should define
TYPE, BIND(C) :: my_data
  REAL, DIMENSION(3) :: x
  REAL :: gap1
  DOUBLE PRECISION :: p
END TYPE
```

Unofficial Draft for Comment Only

 $\overline{7}$

 24

1 2 3 4 5 6	processes in a communication add the same gaps, i.e., defined with the same bas datatype. Both the original and the modified structures are portable, but may hav different performance implications for the communication and memory accesses durin computation on systems with different alignment values.			
7 8 9 10 11	In principle, a compiler may define an additional alignment rule for structures, e.g., to use at least 4 or 8 byte alignment, although the content may have a max_ik_i alignment less than this structure alignment. To maintain portability, users should always resize structure derived datatype handles if used in an array of structures, see the Example in Section 19.1.15. (<i>End of advice to users.</i>)			
12 13 14	5.1.7 Ex	tent and Bounds of Datatype	s	
15 16	MPI_TYPI	E_GET_EXTENT(datatype, lb,	extent)	
17	IN	datatype	datatype to get information on (handle)	
18	OUT	lb	lower bound of datatype (integer)	
19 20	OUT	extent	extent of datatype (integer)	
21				
22	C binding	0		
23	int MPI_7		ype datatype, MPI_Aint *lb,	
24 25		MPI_Aint *extent)		
26	Fortran 2008 binding			
27	<pre>MPI_Type_get_extent(datatype, lb, extent, ierror) TYPE(MPI_Datatype), INTENT(IN) :: datatype</pre>			
28	IYPE(MPI_Datatype), INIENI(IN) :: datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent			
29 30	INTEGER (RIND-MFI_ADDRESS_RIND), INTENT(001) ID, extent INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
31	Fortran k	binding		
32		_GET_EXTENT(DATATYPE, LB,	EXTENT, IERROR)	
33	INTEGER DATATYPE, IERROR			
34 35	INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT			
36				
37	ΜΡΙ ΤΥΡΙ	E_GET_EXTENT_X(datatype,	lb. extent)	
38	- IN	datatype	datatype to get information on (handle)	
39 40	OUT	lb	lower bound of datatype (integer)	
41	OUT	extent	extent of datatype (integer)	
42	001	extent	extent of datatype (integer)	
43	C bindin	g		
44 45	<pre>int MPI_Type_get_extent_x(MPI_Datatype datatype, MPI_Count *1b,</pre>			
45	MPI_Count *extent)			
47	Fortran 2	2008 binding		
48	MPI_Type_	_get_extent_x(datatype, 1	b, extent, ierror)	

<pre>TYPE(MPI_Datatype), INTENT(IN) ::</pre>	datatype		
INTEGER(KIND=MPI_COUNT_KIND), INTE	ENT(OUT) ::	lb,	extent
INTEGER, OPTIONAL, INTENT(OUT) ::	ierror		

Fortran binding

```
MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR)
INTEGER DATATYPE, IERROR
INTEGER(KIND=MPI_COUNT_KIND) LB, EXTENT
```

Returns the lower bound and the extent of datatype (as defined in Equation 5.1). For both functions, if either OUT parameter cannot express the value to be returned (e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED.

MPI allows one to change the extent of a datatype, using lower bound and upper bound markers. This provides control over the stride of successive datatypes that are replicated by datatype constructors, or are replicated by the **count** argument in a send or receive call.

MPI_TYPE_CREATE_RESIZED(oldtype, lb, extent, newtype)

IN	oldtype	input datatype (handle)
IN	lb	new lower bound of datatype (integer)
IN	extent	new extent of datatype (integer)
OUT	newtype	output datatype (handle)

C binding

int	MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb;	,
	MPI_Aint extent, MPI_Datatype *newtype)	

Fortran 2008 binding

<pre>MPI_Type_create_resized(oldtype, lb, extent, newtype,</pre>	ierror)
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	
<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb,</pre>	extent
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	

Fortran binding

```
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
INTEGER OLDTYPE, NEWTYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
```

Returns in **newtype** a handle to a new datatype that is identical to **oldtype**, except that the lower bound of this new datatype is set to be lb, and its upper bound is set to be lb+ extent. Any previous lb and ub markers are erased, and a new pair of lower bound and upper bound markers are put in the positions indicated by the lb and extent arguments. This affects the behavior of the datatype when used in communication operations, with count > 1, and when used in the construction of new derived datatypes.

```
1
     5.1.8 True Extent of Datatypes
\mathbf{2}
     Suppose we implement gather (see also Section 6.5) as a spanning tree implemented on
3
     top of point-to-point routines. Since the receive buffer is only valid on the root pro-
4
     cess, one will need to allocate some temporary space for receiving data on intermedi-
5
     ate nodes. However, the datatype extent cannot be used as an estimate of the amount
6
     of space that needs to be allocated, if the user has modified the extent, for example
7
     by using MPI_TYPE_CREATE_RESIZED. The functions MPI_TYPE_GET_TRUE_EXTENT
8
     and MPI_TYPE_GET_TRUE_EXTENT_X are provided which return the true extent of the
9
     datatype.
10
11
12
     MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)
13
       IN
                 datatype
                                             datatype to get information on (handle)
14
       OUT
                 true_lb
                                             true lower bound of datatype (integer)
15
16
       OUT
                 true_extent
                                             true size of datatype (integer)
17
18
     C binding
19
     int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
20
                    MPI_Aint *true_extent)
21
22
     Fortran 2008 binding
23
     MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
^{24}
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
26
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     Fortran binding
28
     MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
29
          INTEGER DATATYPE, IERROR
30
          INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
^{31}
32
33
     MPI_TYPE_GET_TRUE_EXTENT_X(datatype, true_lb, true_extent)
34
35
       ĬN.
                 datatype
                                             datatype to get information on (handle)
36
       OUT
                 true_lb
                                             true lower bound of datatype (integer)
37
       OUT
                 true_extent
                                             true size of datatype (integer)
38
39
     C binding
40
```

- ⁴⁰ C binding ⁴¹ int MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb, ⁴² MPI_Count *true_extent)
- 43 44 Fortran 2008 binding

```
MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror)
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT true lb returns the effect of the lowest unit of store which is addressed by the detatume

true_lb returns the offset of the lowest unit of store which is addressed by the datatype, i.e., the lower bound of the corresponding typemap, ignoring explicit lower bound markers. true_extent returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring explicit lower bound and upper bound markers, and performing no rounding for alignment. If the typemap associated with datatype is

 $Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$

Then

$$true_lb(Typemap) = min_i \{ disp_i : type_i \neq lb_marker, ub_marker \},$$

$$true_ub(Typemap) = max_i \{ disp_i + sizeof(type_i) : type_i \neq lb_marker, ub_marker \},$$

and

$$true_extent(Typemap) = true_ub(Typemap) - true_lb(typemap).$$

(Readers should compare this with the definitions in Section 5.1.6 and Section 5.1.7, which describe the function MPI_TYPE_GET_EXTENT.)

The true_extent is the minimum number of bytes of memory necessary to hold a datatype, uncompressed.

For both functions, if either OUT parameter cannot express the value to be returned (e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED.

5.1.9 Commit and Free

A datatype object has to be **committed** before it can be used in a communication. As an argument in datatype constructors, uncommitted and also committed datatypes can be used. There is no need to commit basic datatypes. They are "pre-committed."

MPI_TYPE_COMMIT(datatype)		35
INOUT datatype	datatype that is committed (handle)	36
		37
C binding		38
<pre>int MPI_Type_commit(MPI_Datatype *</pre>	datatype)	39 40
		40 41
Fortran 2008 binding		41
MPI_Type_commit(datatype, ierror)		43
TYPE(MPI_Datatype), INTENT(INO INTEGER, OPTIONAL, INTENT(OUT)	01	44
INTEGER, OFITONAL, INTENI(001)	101101	45
Fortran binding		46
MPI_TYPE_COMMIT(DATATYPE, IERROR)		47
INTEGER DATATYPE, IERROR		48

 $\mathbf{2}$

 24

The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed,
 the content of different buffers, with different starting addresses.

Advice to implementors. The system may "compile" at commit time an internal representation for the datatype that facilitates communication, e.g., change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism. (*End of advice to implementors.*)

MPI_TYPE_COMMIT will accept a committed datatype; in this case, it is equivalent to a no-op.

¹⁴ Example 5.10 The following code fragment gives examples of using MPI_TYPE_COMMIT.

```
15
     INTEGER type1, type2
16
     CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
17
                     ! new type object created
18
     CALL MPI_TYPE_COMMIT(type1, ierr)
19
                     ! now type1 can be used for communication
20
     type2 = type1
21
                     ! type2 can be used for communication
22
                     ! (it is a handle to same object as type1)
23
     CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
^{24}
                     ! new uncommitted type object created
25
     CALL MPI_TYPE_COMMIT(type1, ierr)
26
                     ! now type1 can be used anew for communication
27
28
29
     MPI_TYPE_FREE(datatype)
30
^{31}
       INOUT
                datatype
                                           datatype that is freed (handle)
32
33
     C binding
34
     int MPI_Type_free(MPI_Datatype *datatype)
35
     Fortran 2008 binding
36
     MPI_Type_free(datatype, ierror)
37
         TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     Fortran binding
41
     MPI_TYPE_FREE(DATATYPE, IERROR)
42
         INTEGER DATATYPE, IERROR
43
         Marks the datatype object associated with datatype for deallocation and sets datatype
44
45
```

to MPI_DATATYPE_NULL. Any communication that is currently using this datatype will complete normally. Freeing a datatype does not affect any other datatype that was built from the freed datatype. The system behaves as if input datatype arguments to derived datatype constructors are passed by value.

Unofficial Draft for Comment Only

5

6

7

8

9 10

11

5.1. DERIVED DATATYPES

Advice to implementors. The implementation may keep a reference count of active communications that use the datatype, in order to decide when to free it. Also, one may implement constructors of derived datatypes so that they keep pointers to their datatype arguments, rather then copying them. In this case, one needs to keep track of active datatype definition references in order to know when a datatype object can be freed. (*End of advice to implementors.*)

5.1.10 Duplicating a Datatype

MPI_TYPE_DUP(oldtype, newtype)		
IN	oldtype	datatype (handle)
OUT	newtype	copy of oldtype (handle)
C binding int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)		
<pre>Fortran 2008 binding MPI_Type_dup(oldtype, newtype, ierror) TYPE(MPI_Datatype), INTENT(IN) :: oldtype TYPE(MPI_Datatype), INTENT(OUT) :: newtype INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>		
Fortran binding MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR) INTEGER OLDTYPE, NEWTYPE, IERROR		
MPI_TYPE_DUP is a type constructor which duplicates the existing oldtype with as-		

sociated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new datatype. Returns in newtype a new datatype with exactly the same properties as oldtype and any copied cached information, see Section 7.7.4. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded with the functions in Section 5.1.13. The newtype has the same committed state as the old oldtype.

5.1.11 Use of General Datatypes in Communication

Handles to derived datatypes can be passed to a communication call wherever a datatype argument is required. A call of the form MPI_SEND(buf, count, datatype, ...), where count > 1, is interpreted as if the call was passed a new datatype which is the concatenation of count copies of datatype. Thus, MPI_SEND(buf, count, datatype, dest, tag, comm) is equivalent to,

```
MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
MPI_TYPE_COMMIT(newtype)
MPI_SEND(buf, 1, newtype, dest, tag, comm)
MPI_TYPE_FREE(newtype).
```

Similar statements apply to all other communication functions that have a **count** and **datatype** argument.

Suppose that a send operation MPI_SEND(buf, count, datatype, dest, tag, comm) is executed, where datatype has type map,

.

 $\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},\$

and extent *extent*. (Explicit lower bound and upper bound markers are not listed in the type map, but they affect the value of *extent*.) The send operation sends $n \cdot \text{count}$ entries, where entry $i \cdot n + j$ is at location $addr_{i,j} = \text{buf} + extent \cdot i + disp_j$ and has type $type_j$, for $i = 0, \ldots, \text{count} - 1$ and $j = 0, \ldots, n - 1$. These entries need not be contiguous, nor distinct; their order can be arbitrary.

The variable stored at address $addr_{i,j}$ in the calling program should be of a type that matches $type_j$, where type matching is defined as in Section 3.3.1. The message sent contains $n \cdot \text{count entries}$, where entry $i \cdot n + j$ has type $type_j$.

Similarly, suppose that a receive operation MPI_RECV(buf, count, datatype, source, tag, comm, status) is executed, where datatype has type map,

17 18 19

16

 $\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},\$

with extent *extent*. (Again, explicit lower bound and upper bound markers are not listed in the type map, but they affect the value of *extent*.) This receive operation receives $n \cdot \text{count}$ entries, where entry $i \cdot n + j$ is at location $\text{buf} + extent \cdot i + disp_j$ and has type $type_j$. If the incoming message consists of k elements, then we must have $k \leq n \cdot \text{count}$; the $i \cdot n + j$ -th element of the message should have a type that matches $type_j$.

Type matching is defined according to the type signature of the corresponding datatypes, that is, the sequence of basic type components. Type matching does not depend on some aspects of the datatype definition, such as the displacements (layout in memory) or the intermediate types used.

Example 5.11 This example shows that type matching is defined in terms of the basic types that a derived type consists of.

```
32
     . . .
33
     CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, type2, ...)
34
     CALL MPI_TYPE_CONTIGUOUS(4, MPI_REAL, type4, ...)
35
     CALL MPI_TYPE_CONTIGUOUS(2, type2, type22, ...)
36
     . . .
37
     CALL MPI_SEND(a, 4, MPI_REAL, ...)
38
     CALL MPI_SEND(a, 2, type2, ...)
39
     CALL MPI_SEND(a, 1, type22, ...)
40
     CALL MPI_SEND(a, 1, type4, ...)
41
     . . .
42
     CALL MPI_RECV(a, 4, MPI_REAL, ...)
43
     CALL MPI_RECV(a, 2, type2, ...)
^{44}
     CALL MPI_RECV(a, 1, type22, ...)
45
     CALL MPI_RECV(a, 1, type4, ...)
46
     Each of the sends matches any of the receives.
47
48
```

1

 $\mathbf{2}$

3

4

5

6 7

8

9

10

A datatype may specify overlapping entries. The use of such a datatype in a receive operation is erroneous. (This is erroneous even if the actual message received is short enough not to write any entry more than once.)

Suppose that MPI_RECV(buf, count, datatype, dest, tag, comm, status) is executed, where datatype has type map,

```
\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}.
```

The received message need not fill all the receive buffer, nor does it need to fill a number of locations which is a multiple of n. Any number, k, of basic elements can be received, where $0 \le k \le \text{count} \cdot n$. The number of basic elements received can be retrieved from status using the query functions MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X.

MPI_GET_ELEMENTS(status, datatype, count)

IN	status	return status of receive operation (Status)
IN	datatype	datatype used by receive operation (handle)
OUT	count	number of received basic elements (integer)

C binding

```
Fortran 2008 binding24MPI_Get_elements(status, datatype, count, ierror)25TYPE(MPI_Status), INTENT(IN) :: status26TYPE(MPI_Datatype), INTENT(IN) :: datatype27INTEGER, INTENT(OUT) :: count28INTEGER, OPTIONAL, INTENT(OUT) :: ierror29
```

Fortran binding

```
MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

MPI_GET_ELEMENTS_X(status, datatype, count)

IN	status	return status of receive operation (Status)
IN	datatype	datatype used by receive operation (handle)
OUT	count	number of received basic elements (integer)

C binding

Fortran 2008 binding

```
MPI_Get_elements_x(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(IN) :: status
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

 $\mathbf{2}$

1	INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
3	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4	Fortran binding
5	MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
6	INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
7	INTEGER(KIND=MPI_COUNT_KIND) COUNT
8	The datatype argument should match the argument provided by the receive call that
9	set the status variable. For both functions, if the OUT parameter cannot express the value
10	to be returned (e.g., if the parameter is too small to hold the output value), it is set to
11	MPI_UNDEFINED.
12	The previously defined function MPI_GET_COUNT (Section 3.2.5), has a different be-
13	havior. It returns the number of "top-level entries" received, i.e. the number of "copies" of
14	type datatype. In the previous example, MPI_GET_COUNT may return any integer value
15	k, where $0 \le k \le \text{count}$. If MPI_GET_COUNT returns k, then the number of basic elements
16	received (and the value returned by MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X) is
17	$n \cdot k$. If the number of basic elements received is not a multiple of n, that is, if the receive
18	operation has not received an integral number of datatype "copies," then MPI_GET_COUNT
19	sets the value of count to MPI_UNDEFINED.
20	
21	Example 5.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.
22	
23	CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
24 25	CALL MPI_TYPE_COMMIT(Type2, ierr)
25 26	
20	CALL MPI_COMM_RANK(comm, rank, ierr)
28	IF (rank.EQ.0) THEN
29	CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
30	CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
31	ELSE IF (rank.EQ.1) THEN
32	CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
33	CALL MPI_GET_COUNT(stat, Type2, i, ierr) ! returns i=1
34	CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) ! returns i=2
35	CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
36	CALL MPI_GET_COUNT(stat, Type2, i, ierr) ! returns i=MPI_UNDEFINED
37	CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) ! returns i=3 END IF
38	
39	The functions MPI_GET_ELEMENTS and MPI_GET_ELEMENTS_X can also be used
40	after a probe to find the number of elements in the probed message. Note that the
41	MPI_GET_COUNT, MPI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X return the same
42	values when they are used with basic datatypes as long as the limits of their respective
43	count arguments are not exceeded.
44 45	
45 46	<i>Rationale.</i> The extension given to the definition of MPI_GET_COUNT seems natural:
40	one would expect this function to return the value of the count argument, when the
48	receive buffer is filled. Sometimes datatype represents a basic unit of data one wants to transfer, for example, a record in an array of records (structures). One should be
	to transfer, for example, a record in an array of records (structures). One should be

Unofficial Draft for Comment Only

able to find out how many components were received without bothering to divide by the number of elements in each component. However, on other occasions, datatype is used to define a complex layout of data in the receiver memory, and does not represent a basic unit of data for transfers. In such cases, one needs to use the function MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X. (*End of rationale.*)

Advice to implementors. The definition implies that a receive cannot change the value of storage outside the entries defined to compose the communication buffer. In particular, the definition implies that padding space in a structure should not be modified when such a structure is copied from one process to another. This would prevent the obvious optimization of copying the structure, together with the padding, as one contiguous block. The implementation is free to do this optimization when it does not impact the outcome of the computation. The user can "force" this optimization by explicitly including padding as part of the message. (End of advice to implementors.)

5.1.12 Correct Use of Addresses

Successively declared variables in C or Fortran are not necessarily stored at contiguous locations. Thus, care must be exercised that displacements do not cross from one variable to another. Also, in machines with a segmented address space, addresses are not unique and address arithmetic has some peculiar properties. Thus, the use of **addresses**, that is, displacements relative to the start address MPI_BOTTOM, has to be restricted.

Variables belong to the same **sequential storage** if they belong to the same array, to the same COMMON block in Fortran, or to the same structure in C. Valid addresses are defined recursively as follows:

- 1. The function MPI_GET_ADDRESS returns a valid address, when passed as argument a variable of the calling program.
- 2. The **buf** argument of a communication function evaluates to a valid address, when passed as argument a variable of the calling program.
- 3. If v is a valid address, and i is an integer, then v+i is a valid address, provided v and v+i are in the same sequential storage.

A correct program uses only valid addresses to identify the locations of entries in communication buffers. Furthermore, if u and v are two valid addresses, then the (integer) difference u - v can be computed only if both u and v are in the same sequential storage. No other arithmetic operations can be meaningfully Aexecuted on addresses.

The rules above impose no constraints on the use of derived datatypes, as long as they are used to define a communication buffer that is wholly contained within the same sequential storage. However, the construction of a communication buffer that contains variables that are not within the same sequential storage must obey certain restrictions. Basically, a communication buffer with variables that are not within the same sequential storage can be used only by specifying in the communication call buf = MPI_BOTTOM, count = 1, and using a datatype argument where all displacements are valid (absolute) addresses.

Advice to users.It is not expected that MPI implementations will be able to detect45erroneous, "out of bound" displacements — unless those overflow the user address46space — since the MPI call may not know the extent of the arrays and records in the47host program.(End of advice to users.)48

Unofficial Draft for Comment Only

 $\overline{7}$

 24

 31

Advice to implementors. There is no need to distinguish (absolute) addresses and (relative) displacements on a machine with contiguous address space: MPI_BOTTOM is zero, and both addresses and displacements are integers. On machines where the distinction is required, addresses are recognized as expressions that involve MPI_BOTTOM. (*End of advice to implementors.*)

5.1.13 Decoding a Datatype

MPI datatype objects allow users to specify an arbitrary layout of data in memory. There are several cases where accessing the layout information in opaque datatype objects would be useful. The opaque datatype object has found a number of uses outside MPI. Furthermore, a number of tools wish to display internal information about a datatype. To achieve this, datatype decoding functions are provided. The two functions in this section are used together to decode datatypes to recreate the calling sequence used in their initial definition. These can be used to allow a user to determine the type map and type signature of a datatype.

18 MPI_TYPE_GET_ENVELOPE(datatype, num_integers, num_addresses, num_datatypes, 19 combiner) 20IN datatype to access (handle) datatype 2122 OUT number of input integers used in call constructing num_integers 23combiner (non-negative integer) 24OUT num_addresses number of input addresses used in call constructing 25combiner (non-negative integer) 26OUT num_datatypes number of input datatypes used in call constructing 27combiner (non-negative integer) 2829 OUT combiner combiner (state) 30 31 C binding 32 int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers, 33 int *num_addresses, int *num_datatypes, int *combiner) 34 Fortran 2008 binding 35 MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes, 36 combiner, ierror) 37 TYPE(MPI_Datatype), INTENT(IN) :: datatype 38 INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes, 39 combiner 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 42Fortran binding 43 MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, 44 COMBINER, IERROR) 45 INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER, 46 IERROR 47 48

1

2

3

4

5

6 7

8

9

10

11

12

13

14

15

For the given datatype, MPI_TYPE_GET_ENVELOPE returns information on the number and type of input arguments used in the call that created the datatype. The number-ofarguments values returned can be used to provide sufficiently large arrays in the decoding routine MPI_TYPE_GET_CONTENTS. This call and the meaning of the returned values is described below. The combiner reflects the MPI datatype constructor call that was used in creating datatype.

Rationale. By requiring that the combiner reflect the constructor used in the creation of the datatype, the decoded information can be used to effectively recreate the calling sequence used in the original creation. This is the most useful information and was felt to be reasonable even though it constrains implementations to remember the original constructor sequence even if the internal representation is different.

The decoded information keeps track of datatype duplications. This is important as one needs to distinguish between a predefined datatype and a dup of a predefined datatype. The former is a constant object that cannot be freed, while the latter is a derived datatype that can be freed. (End of rationale.)

The list in Table 5.1 has the values that can be returned in combiner on the left and the call associated with them on the right.

MPI_COMBINER_NAMED	a named predefined datatype	21
MPI_COMBINER_DUP	MPI_TYPE_DUP	22
MPI_COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS	23
MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR	24
MPI_COMBINER_HVECTOR	MPI_TYPE_CREATE_HVECTOR	25
MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED	26
MPI_COMBINER_HINDEXED	MPI_TYPE_CREATE_HINDEXED	27
MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK	28
MPI_COMBINER_HINDEXED_BLOCK	MPI_TYPE_CREATE_HINDEXED_BLOCK	29
MPI_COMBINER_STRUCT	MPI_TYPE_CREATE_STRUCT	30
MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY	31
MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY	32
MPI_COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL	33
MPI_COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX	34
MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER	35
MPI_COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED	36
		37
Table 5.1. combiner values return	ed from MPI_TYPE_GET_ENVELOPE	38
Table 5.1. combiner values return		39
If combiner is MPI_COMBINER_NAMED) then datatype is a named predefined datatype.	40
		41
MPI_TYPE_GET_CONTENTS.		42
		43
		44

1 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

7

8

9

10

11

1213

14

15

16

17 18

19

20

```
1
     MPI_TYPE_GET_CONTENTS(datatype, max_integers, max_addresses, max_datatypes,
\mathbf{2}
                    array_of_integers, array_of_addresses, array_of_datatypes)
3
       IN
                 datatype
                                             datatype to access (handle)
4
       IN
                 max_integers
                                             number of elements in array_of_integers
5
                                             (non-negative integer)
6
7
       IN
                 max_addresses
                                             number of elements in array_of_addresses
8
                                             (non-negative integer)
9
       IN
                 max_datatypes
                                             number of elements in array_of_datatypes
10
                                             (non-negative integer)
11
       OUT
                 array_of_integers
                                             contains integer arguments used in constructing
12
                                             datatype (array of integers)
13
       OUT
                                             contains address arguments used in constructing
14
                 array_of_addresses
                                             datatype (array of integers)
15
16
       OUT
                                             contains datatype arguments used in constructing
                 array_of_datatypes
17
                                             datatype (array of handles)
18
19
     C binding
20
     int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
21
                    int max_addresses, int max_datatypes, int array_of_integers[],
22
                    MPI_Aint array_of_addresses[],
23
                    MPI_Datatype array_of_datatypes[])
^{24}
25
     Fortran 2008 binding
26
     MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,
27
                    array_of_integers, array_of_addresses, array_of_datatypes,
                    ierror)
28
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
          INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes
30
          INTEGER, INTENT(OUT) :: array_of_integers(max_integers)
31
32
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
33
                     array_of_addresses(max_addresses)
34
          TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     Fortran binding
37
     MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
38
                    ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,
39
                    IERROR)
40
          INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
41
                     ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR
42
          INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)
43
44
          datatype must be a predefined unnamed or a derived datatype; the call is erroneous if
45
     datatype is a predefined named datatype.
46
          The values given for max_integers, max_addresses, and max_datatypes must be at least as
47
     large as the value returned in num_integers, num_addresses, and num_datatypes, respectively,
48
     in the call MPI_TYPE_GET_ENVELOPE for the same datatype argument.
```

CHAPTER 5. DATATYPES

Rationale. The arguments max_integers, max_addresses, and max_datatypes allow for error checking in the call. (*End of rationale.*)

The datatypes returned in array_of_datatypes are handles to datatype objects that are equivalent to the datatypes used in the original construction call. If these were derived datatypes, then the returned datatypes are new datatype objects, and the user is responsible for freeing these datatypes with MPI_TYPE_FREE. If these were predefined datatypes, then the returned datatype is equal to that (constant) predefined datatype and cannot be freed.

The committed state of returned derived datatypes is undefined, i.e., the datatypes may or may not be committed. Furthermore, the content of attributes of returned datatypes is undefined.

Note that MPI_TYPE_GET_CONTENTS can be invoked with a datatype argument that was constructed using MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_INTEGER, or MPI_TYPE_CREATE_F90_COMPLEX (an unnamed predefined datatype). In such a case, an empty array_of_datatypes is returned.

Rationale. The definition of datatype equivalence implies that equivalent predefined datatypes are equal. By requiring the same handle for named predefined datatypes, it is possible to use the == or .EQ. comparison operator to determine the datatype involved. (*End of rationale.*)

Advice to implementors. The datatypes returned in array_of_datatypes must appear to the user as if each is an equivalent copy of the datatype used in the type constructor call. Whether this is done by creating a new datatype or via another mechanism such as a reference count mechanism is up to the implementation as long as the semantics are preserved. (*End of advice to implementors.*)

Rationale. The committed state and attributes of the returned datatype is deliberately left vague. The datatype used in the original construction may have been modified since its use in the constructor call. Attributes can be added, removed, or modified as well as having the datatype committed. The semantics given allow for a reference count implementation without having to track these changes. (*End of rationale.*)

In the deprecated datatype constructor calls, the address arguments in Fortran are of type INTEGER. In the preferred calls, the address arguments are of type INTEGER(KIND=MPI_ADDRESS_KIND). The call MPI_TYPE_GET_CONTENTS returns all addresses in an argument of type INTEGER(KIND=MPI_ADDRESS_KIND). This is true even if the deprecated calls were used. Thus, the location of values returned can be thought of as being returned by the C bindings. It can also be determined by examining the preferred calls for datatype constructors for the deprecated calls that involve addresses.

Rationale. By having all address arguments returned in the array_of_addresses argument, the result from a C and Fortran decoding of a datatype gives the result in the same argument. It is assumed that an integer of type INTEGER(KIND=MPI_ADDRESS_KIND) will be at least as large as the INTEGER argument used in datatype construction with the old MPI-1 calls so no loss of information will occur. (End of rationale.)

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11

12

13

14

15 16

17

18

19

20 21

22

23

 24

25

26 27

28

29

30

 31

32

33 34

35

36

37

38

39

40

 $41 \\ 42$

43

44

45

46

47

1 The following defines what values are placed in each entry of the returned arrays $\mathbf{2}$ depending on the datatype constructor used for datatype. It also specifies the size of the 3 arrays needed which is the values returned by MPI_TYPE_GET_ENVELOPE. In Fortran, 4 the following calls were made: 5PARAMETER (LARGE = 1000) 6 INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR 7 INTEGER (KIND=MPI_ADDRESS_KIND) A(LARGE) 8 ! CONSTRUCT DATATYPE TYPE (NOT SHOWN) 9 CALL MPI_TYPE_GET_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR) 10 IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN 11 WRITE (*, *) "NI, NA, OR ND = ", NI, NA, ND, & 12" RETURNED BY MPI_TYPE_GET_ENVELOPE IS LARGER THAN LARGE = ", LARGE 13 CALL MPI_ABORT(MPI_COMM_WORLD, 99, IERROR) 14ENDIF 15CALL MPI_TYPE_GET_CONTENTS(TYPE, NI, NA, ND, I, A, D, IERROR) 1617 or in C the analogous calls of: 18 19#define LARGE 1000 20int ni, na, nd, combiner, i[LARGE]; 21MPI_Aint a[LARGE]; 22MPI_Datatype type, d[LARGE]; 23/* construct datatype type (not shown) */ 24 MPI_Type_get_envelope(type, &ni, &na, &nd, &combiner); 25if ((ni > LARGE) || (na > LARGE) || (nd > LARGE)) { 26fprintf(stderr, "ni, na, or nd = %d %d %d returned by ", ni, na, nd); 27fprintf(stderr, "MPI_Type_get_envelope is larger than LARGE = %d\n", 28LARGE); 29 MPI_Abort(MPI_COMM_WORLD, 99); 30 }; 31MPI_Type_get_contents(type, ni, na, nd, i, a, d); 32 In the descriptions that follow, the lower case name of arguments is used. 33 If combiner is MPI_COMBINER_NAMED then it is erroneous to call 34 MPI_TYPE_GET_CONTENTS. 35If combiner is MPI_COMBINER_DUP then 36 37 Constructor argument С Fortran location 38 oldtype d[0]D(1)39 40and ni = 0, na = 0, nd = 1. 41 If combiner is MPI_COMBINER_CONTIGUOUS then 4243 Constructor argument С Fortran location 44 i[0]I(1)count 45d[0]D(1)oldtype 46and ni = 1, na = 0, nd = 1. 47 If combiner is MPI_COMBINER_VECTOR then 48

and

and

and

and

and

oldtype

	Constructor argumen	t C Fort	ran location	
	count	i[0]	I(1)	
	blocklength	i[1]	I(2)	
	stride	i[2]	I(3)	
	oldtype	d[0]	D(1)	
1 • 0	0 1 1			
	na = 0, nd = 1.			
II combi	ner is MPI_COMBINER_HVECT	JR then		
	Constructor argumen	t C Fort	ran location	
	count	i[0]	I(1)	
	blocklength	i[1]	I(2)	
	stride	a[0]	A(1)	
	oldtype	d[0]	D(1)	
,	na = 1, nd = 1.			
If combi	ner is MPI_COMBINER_INDEXE	D then		
Cor	nstructor argument	С	Fortran location	
cou		i[0]	I(1)	L
		to $i[i[0]]$	I(1) I(2) to $I(I(1)+1)$)
			I(1) + 2 to $I(1) + 1I(I(1) + 2)$ to $I(2*I(1))$	
arr			1(1) - 2 = 0 - 1(2) + 0 - 1(2) + 0 - 1(2) T'I)
			$\mathbf{D}(1)$	
		d[0]	D(1)	
old			D(1)	
$-\frac{\text{old}}{\text{d}}$ d ni = 2*c	type	d[0]	D(1)	
$-\frac{\text{old}}{\text{d}}$ d ni = 2*c	type count+1, $na = 0$, $nd = 1$. ner is MPI_COMBINER_HINDEX	d[0]		
$-\frac{\text{old}}{\text{d}}$ d ni = 2*c	type count+1, $na = 0$, $nd = 1$. ner is MPI_COMBINER_HINDEX Constructor argument	d[0] C	Fortran location	
$-\frac{\text{old}}{\text{d}}$ d ni = 2*c	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count	$\frac{d[0]}{\text{CD then}}$	Fortran location I(1)	
$-\frac{\text{old}}{\text{d}}$ d ni = 2*c	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths	$\frac{d[0]}{\text{ED then}}$ $\frac{C}{i[0]}$ $i[1] \text{ to } i[i[0]]$	Fortran location I(1) I(2) to I(I(1)+1)	
$-\frac{\text{old}}{\text{d}}$ d ni = 2*c	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements a	$\frac{d[0]}{C}$ CED then $\frac{C}{i[0]}$ i[1] to i[i[0]] [0] to a[i[0]-1]	Fortran location I(1)	
$-\frac{\text{old}}{\text{d}}$ d ni = 2*c	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths	$\frac{d[0]}{\text{ED then}}$ $\frac{C}{i[0]}$ $i[1] \text{ to } i[i[0]]$	Fortran location I(1) I(2) to I(I(1)+1)	
old d ni = 2*c If combi	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements al oldtype	$\frac{d[0]}{C}$ CED then $\frac{C}{i[0]}$ i[1] to i[i[0]] [0] to a[i[0]-1]	Fortran location I(1) I(2) to I(I(1)+1) A(1) to A(I(1))	
old d ni = 2*c If combi d ni = cou	type type type total count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements a oldtype unt+1, na = count, nd = 1.	$\frac{d[0]}{C}$ ED then $\frac{C}{i[0]}$ i[1] to i[i[0]] [0] to a[i[0]-1] d[0]	Fortran location I(1) I(2) to $I(I(1)+1)A(1)$ to $A(I(1))D(1)$	
old d ni = 2*c If combi d ni = cou	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements al oldtype	$\frac{d[0]}{C}$ ED then $\frac{C}{i[0]}$ i[1] to i[i[0]] [0] to a[i[0]-1] d[0]	Fortran location I(1) I(2) to $I(I(1)+1)A(1)$ to $A(I(1))D(1)$	
old d ni = 2*c If combi d ni = cou	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements al oldtype unt+1, na = count, nd = 1. ner is MPI_COMBINER_INDEXE	$\frac{d[0]}{C}$ ED then $\frac{C}{i[0]}$ i[1] to i[i[0]] [0] to a[i[0]-1] d[0]	Fortran location I(1) I(2) to $I(I(1)+1)A(1)$ to $A(I(1))D(1)$	
old d ni = 2*c If combi d ni = cou	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements a oldtype unt+1, na = count, nd = 1. ner is MPI_COMBINER_INDEXE Constructor argument	d[0] C i[0] i[1] to i[i[0]] [0] to a[i[0]-1] d[0]	Fortran location I(1) I(2) to I(I(1)+1) A(1) to A(I(1)) D(1) n Fortran location	
old d ni = 2*c If combi d ni = cou	type type type total: type total: type	$\frac{d[0]}{C}$ C then $\frac{C}{i[0]}$ i[1] to i[i[0]] i[0] to a[i[0]-1] d[0] C to BLOCK the $\frac{C}{i[0]}$	Fortran location I(1) $I(2) to I(I(1)+1)$ $A(1) to A(I(1))$ $D(1)$ n Fortran location $I(1)$	
old d ni = 2*c If combi d ni = cou	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements al oldtype unt+1, na = count, nd = 1. ner is MPI_COMBINER_INDEXE Constructor argument count blocklength	$\frac{d[0]}{C} \\ C \\ i[0] \\ i[1] to i[i[0]] \\ i[1] to a[i[0]-1] \\ d[0] \\ C \\ C \\ i[0] \\ i[1] \\ i[1] \\ c \\ c \\ c \\ c \\ c \\ c \\ i[0] \\ i[1] \\ c \\ $	Fortran location I(1) I(2) to I(I(1)+1) A(1) to A(I(1)) D(1) n Fortran location I(1) I(2)	
old d ni = 2*c If combi d ni = cou	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements al oldtype unt+1, na = count, nd = 1. ner is MPI_COMBINER_INDEXE Constructor argument count blocklength array_of_displacements i[$\frac{d[0]}{C}$ CED then $\frac{C}{i[0]}$ i[1] to i[i[0]] [0] to a[i[0]-1] d[0] C D_BLOCK the $\frac{C}{i[0]}$ i[1] 2] to i[i[0]+1]	Fortran location I(1) $I(2) to I(I(1)+1)$ $A(1) to A(I(1))$ $D(1)$ n Fortran location $I(1)$ $I(2)$ $I(3) to I(I(1)+2)$	
old d ni = 2*c If combi d ni = cou	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements al oldtype unt+1, na = count, nd = 1. ner is MPI_COMBINER_INDEXE Constructor argument count blocklength	$\frac{d[0]}{C} \\ C \\ i[0] \\ i[1] to i[i[0]] \\ i[1] to a[i[0]-1] \\ d[0] \\ C \\ C \\ i[0] \\ i[1] \\ i[1] \\ c \\ c \\ c \\ c \\ c \\ c \\ i[0] \\ i[1] \\ c \\ $	Fortran location I(1) I(2) to I(I(1)+1) A(1) to A(I(1)) D(1) n Fortran location I(1) I(2)	
old d ni = 2*c If combi d ni = cou If combi d ni = cou	type type type total: type total: type	$\frac{d[0]}{C}$ CED then $\frac{C}{i[0]}$ i[1] to i[i[0]] [0] to a[i[0]-1] d[0] CD_BLOCK the $\frac{C}{i[0]}$ i[1] 2] to i[i[0]+1] d[0] C	Fortran location I(1) I(2) to I(I(1)+1) A(1) to A(I(1)) D(1) n Fortran location I(1) I(2) I(3) to I(I(1)+2) D(1)	
old d ni = 2*c If combi d ni = cou If combi d ni = cou	type type type tount+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements a oldtype unt+1, na = count, nd = 1. ner is MPI_COMBINER_INDEXE Constructor argument count blocklength array_of_displacements i[soldtype]	$\frac{d[0]}{C}$ CED then $\frac{C}{i[0]}$ i[1] to i[i[0]] [0] to a[i[0]-1] d[0] CD_BLOCK the $\frac{C}{i[0]}$ i[1] 2] to i[i[0]+1] d[0] C	Fortran location I(1) I(2) to I(I(1)+1) A(1) to A(I(1)) D(1) n Fortran location I(1) I(2) I(3) to I(I(1)+2) D(1)	
old d ni = 2*c If combi d ni = cou If combi d ni = cou	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements a oldtype mt+1, na = count, nd = 1. ner is MPI_COMBINER_INDEXE Constructor argument count blocklength array_of_displacements i[: oldtype mt+2, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX	$\frac{d[0]}{C}$ CED then $\frac{C}{i[0]}$ i[1] to i[i[0]] [0] to a[i[0]-1] d[0] CD_BLOCK the $\frac{C}{i[0]}$ i[1] 2] to i[i[0]+1] d[0] CED_BLOCK the $\frac{C}{C}$	Fortran location I(1) I(2) to I(I(1)+1) A(1) to A(I(1)) D(1) n Fortran location I(1) I(2) I(3) to I(I(1)+2) D(1) en	
old d ni = 2*c If combi d ni = cou If combi d ni = cou	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements a oldtype unt+1, na = count, nd = 1. ner is MPI_COMBINER_INDEXE Constructor argument count blocklength array_of_displacements i[sound oldtype unt+2, na = 0, nd = 1. ner is MPI_COMBINER_HINDEXE	$\frac{d[0]}{C}$ CD then $\frac{C}{i[0]}$ i[1] to i[i[0]] [0] to a[i[0]-1] d[0] CD_BLOCK the $\frac{C}{i[0]}$ i[1] 2] to i[i[0]+1] d[0] C C C C	Fortran location I(1) I(2) to I(I(1)+1) A(1) to A(I(1)) D(1) n Fortran location I(1) I(2) I(3) to I(I(1)+2) D(1) en Fortran location	
old d ni = 2*c If combi d ni = cou If combi d ni = cou	type type type tount+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements al oldtype tout+1, na = count, nd = 1. ner is MPI_COMBINER_INDEXE Constructor argument count blocklength array_of_displacements i[sount+2, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count count	$\frac{d[0]}{C} = \frac{C}{i[0]}$ $\frac{C}{i[1] \text{ to } i[i[0]]} = \frac{C}{i[0]}$ $\frac{C}{i[0]} = \frac{C}{i[0]} = \frac{C}{i[1]}$ $\frac{C}{i[0]} = \frac{C}{i[0]} = \frac{C}{i[0]} = \frac{C}{i[0]}$ $\frac{C}{i[0]} = \frac{C}{i[0]} = $	Fortran location I(1) I(2) to I(I(1)+1) A(1) to A(I(1)) D(1) n Fortran location I(1) I(2) I(3) to I(I(1)+2) D(1) en Fortran location I(1)	
old d ni = 2*c If combi d ni = cou If combi d ni = cou	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements a oldtype unt+1, na = count, nd = 1. ner is MPI_COMBINER_INDEXE Constructor argument count blocklength array_of_displacements i[sound oldtype unt+2, na = 0, nd = 1. ner is MPI_COMBINER_HINDEXE	$\frac{d[0]}{C}$ CD then $\frac{C}{i[0]}$ i[1] to i[i[0]] [0] to a[i[0]-1] d[0] CD_BLOCK the $\frac{C}{i[0]}$ i[1] 2] to i[i[0]+1] d[0] C C C C	Fortran location I(1) I(2) to I(I(1)+1) A(1) to A(I(1)) D(1) n Fortran location I(1) I(2) I(3) to I(I(1)+2) D(1) en Fortran location	
old d ni = 2*c If combi d ni = cou If combi d ni = cou	type count+1, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count array_of_blocklengths array_of_displacements al oldtype mt+1, na = count, nd = 1. ner is MPI_COMBINER_INDEXE Constructor argument count blocklength array_of_displacements i[: oldtype mt+2, na = 0, nd = 1. ner is MPI_COMBINER_HINDEX Constructor argument count blocklength array_of_displacements i[: oldtype	$\frac{d[0]}{C} = \frac{C}{i[0]}$ $\frac{C}{i[1] \text{ to } i[i[0]]} = \frac{C}{i[0]}$ $\frac{C}{i[0]} = \frac{C}{i[0]} = \frac{C}{i[1]}$ $\frac{C}{i[0]} = \frac{C}{i[0]} = \frac{C}{i[0]} = \frac{C}{i[0]}$ $\frac{C}{i[0]} = \frac{C}{i[0]} = $	Fortran location I(1) I(2) to I(I(1)+1) A(1) to A(I(1)) D(1) n Fortran location I(1) I(2) I(3) to I(I(1)+2) D(1) en Fortran location I(1) I(2)	

Unofficial Draft for Comment Only

d[0]

D(1)

/	= count, nd $=$ 1.			
	er is MPI_COMBINER_	STRUCT then		
11 0011101110				
-	Constructor argume	ent C	Fortran locati	on
-	count	i[0]	I(1)	
	array_of_blocklengt	hs i[1] to i[i[0] I(2) to I(I(1)+	-1)
	array_of_displaceme		[0]-1] A(1) to A(I(1))))
	array_of_types	d[0] to $d[i]$	I[0]-1] D(1) to $D(I(1)$))
-				
	t+1, na = count, nd			
II combine	er is MPI_COMBINER_	SUBARRAY the	n	
Constr	uctor argument	С	Fortran loca	ation
ndims		i[0]	I(1)	
	of_sizes	i[1] to i[i[0]]	I(1) I(2) to I(I(1)	$) \pm 1)$
•		i[0]+1] to $i[2*i[0]$		
e e	Ľ	i[0]+1] to $i[2, i]0i[0]+1]$ to $i[3*i]$		
order	1[2	i[3*i[0]+1] to $i[3*i[0]+1]$	I(2 I(1)+2) to I(1)+1(2) I(3*I(1)+1) I(3	
oldtype	د د	d[0]	D(1)	2]
Oldtypt		u[0]		
and $ni = 3*ndi$	ims+2, $na = 0$, $nd =$	= 1.		
If $combine$	er is MPI_COMBINER_	DARRAY then		
	ctor argument	С	Fortran lo	cation
size		i[0]	I(1)	
rank		i[1]	I(2)	
ndims		i[2]	I(3)	
array_of	0	i[3] to i[i[2]+2]	I(4) to $I(I)$	· / /
v		2]+3] to $i[2*i[2]-$,
array_of	_	[2]+3] to $i[3*i[2]$		
array_of	_psizes i[3*i	[2]+3] to $i[4*i[2]$		
order		i[4*i[2]+3]	I(4*I(3))	+4)
oldtype		d[0]	D(1))
1 • • • •		1		
	ims+4, $na = 0$, $nd =$			
If combine	er is MPI_COMBINER_	F90_REAL then		
	Constructor a	argument C	Fortran location	
		i[0]	I(1)	
	p r		I(1) I(2)	
	r	i[1]	1(2)	
	= 0, nd = 0.			
and $ni = 2$, na	,	F90_COMPLEX	then	
,	T IS MPI_COMBINER			
,	er is MPI_COMBINER			
,	Constructor	argument C	Fortran location	
,		argument C i[0]	Fortran location I(1)	
,	Constructor a	0		
,	Constructor a p r	i[0]	I(1)	

	Constructor argument	С	Fortran location
	r	i[0]	I(1)
ni = 1, na = 0 If combiner is	0, nd = 0. MPI_COMBINER_RESIZED	then	
	Constructor argument	С	Fortran location
	lb	a[0]	A(1)
	extent	a[1]	A(2)
	oldtype	d[0]	D(1)
d ni = 0, na = 2	2, nd = 1.		
14 Examples			
	mples illustrate the use of d	lorivor	dataturas
le following exal	inples inustrate the use of c	lerived	r datatypes.
ample 5.13 S	end and receive a section o	f a 3I) array.
AT (100 100			
	100), e(9,9,9) e, twoslice, threeslice	. m	conk iorr
	PI_ADDRESS_KIND) lb, si	•	
	MPI_STATUS_SIZE)		
extract the s	ection a(1:17:2, 3:11,	2:10)	
and store it	in e(:,:,:).		
LL MPI_COMM_R.	ANK(MPI_COMM_WORLD, myr	rank,	lerr)
NI MPT TYPF C	ET_EXTENT(MPI_REAL, 1b,	si70	ofreal ierr)
	DI_DAIDNI (III I_IUAD, IO,	(D12)	, ieii)
create dataty	pe for a 1D section		
	ECTOR(9, 1, 2, MPI_REAI	., one	eslice, ierr)
	pe for a 2D section		
LL MPI_TYPE_C	REATE_HVECTOR(9, 1, 100		
	two	oslice	e, ierr)
create datato	pe for the entire secti	on	
• .	REATE_HVECTOR(9, 1, 100		ksizeofreal two
	threeslic		
		.,	-
LL MPI_TYPE_C	OMMIT(threeslice, ierr))	
LL MPI_SENDRE	CV(a(1,3,2), 1, threes]	ice,	myrank, 0, e, 9
	MPI_REAL, myrank, 0,	MPI.	_COMM_WORLD, sta
xample 5.14 (Copy the (strictly) lower tri	anguls	ar part of a matrix
	Copy the (Strictly) lower th		- Poirt or a mattin

```
1
     REAL a(100,100), b(100,100)
\mathbf{2}
     INTEGER disp(100), blocklen(100), ltype, myrank, ierr
3
     INTEGER status(MPI_STATUS_SIZE)
4
\mathbf{5}
     ! copy lower triangular part of array a
6
     ! onto lower triangular part of array b
7
8
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
9
10
     ! compute start and size of each column
^{11}
     DO i=1,100
12
        disp(i) = 100*(i-1) + i
13
        blocklen(i) = 100-i
14
     END DO
15
16
     ! create datatype for lower triangular part
17
     CALL MPI_TYPE_INDEXED(100, blocklen, disp, MPI_REAL, ltype, ierr)
18
19
     CALL MPI_TYPE_COMMIT(ltype, ierr)
20
     CALL MPI_SENDRECV(a, 1, ltype, myrank, 0, b, 1, &
21
                        ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)
22
23
     Example 5.15 Transpose a matrix.
24
     REAL a(100,100), b(100,100)
25
     INTEGER row, xpose, myrank, ierr
26
     INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
27
     INTEGER status(MPI_STATUS_SIZE)
28
29
     ! transpose matrix a onto b
30
31
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
32
33
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
34
35
     ! create datatype for one row
36
     CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)
37
38
     ! create datatype for matrix in row-major order
39
     CALL MPI_TYPE_CREATE_HVECTOR(100, 1, sizeofreal, row, xpose, ierr)
40
41
     CALL MPI_TYPE_COMMIT(xpose, ierr)
42
43
     ! send matrix in row-major order and receive in column major order
44
     CALL MPI_SENDRECV(a, 1, xpose, myrank, 0, b, 100*100, &
45
                        MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
46
47
48
     Example 5.16 Another approach to the transpose problem:
```

```
1
REAL a(100,100), b(100,100)
                                                                                      \mathbf{2}
INTEGER row, row1
                                                                                      3
INTEGER (KIND=MPI_ADDRESS_KIND) disp(2), lb, sizeofreal
INTEGER myrank, ierr
                                                                                      4
INTEGER status (MPI_STATUS_SIZE)
                                                                                      5
                                                                                      6
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
                                                                                      9
! transpose matrix a onto b
                                                                                      10
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
                                                                                      11
                                                                                      12
! create datatype for one row
                                                                                      13
                                                                                      14
CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)
                                                                                      15
                                                                                      16
! create datatype for one row, with the extent of one real number
                                                                                      17
1b = 0
CALL MPI_TYPE_CREATE_RESIZED(row, lb, sizeofreal, row1, ierr)
                                                                                      18
                                                                                      19
CALL MPI_TYPE_COMMIT(row1, ierr)
                                                                                      20
                                                                                      21
! send 100 rows and receive in column major order
                                                                                      22
CALL MPI_SENDRECV(a, 100, row1, myrank, 0, b, 100*100, &
                                                                                      23
                                                                                      ^{24}
                   MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
                                                                                      25
                                                                                      26
Example 5.17 Use of MPI datatypes to manipulate an array of structures.
                                                                                      27
struct Partstruct
                                                                                      28
                                                                                      29
{
                                                                                      30
           type; /* particle type */
   int
                                                                                      31
   double d[6];
                   /* particle coordinates */
                   /* some additional information */
                                                                                      32
   char
          b[7];
                                                                                      33
};
                                                                                      34
                                                                                      35
struct Partstruct
                       particle[1000];
                                                                                      36
                                                                                      37
int
              i, dest, tag;
                                                                                      38
MPI_Comm
              comm;
                                                                                      39
                                                                                      40
                                                                                      41
/* build datatype describing structure */
                                                                                      42
MPI_Datatype Particlestruct, Particletype;
                                                                                      43
                                                                                      44
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
              blocklen[3] = \{1, 6, 7\};
                                                                                      45
int
                                                                                      46
MPI_Aint
              disp[3];
                                                                                      47
MPI_Aint
              base, lb, sizeofentry;
                                                                                      48
```

```
1
\mathbf{2}
     /* compute displacements of structure components */
3
4
     MPI_Get_address(particle, disp);
5
     MPI_Get_address(particle[0].d, disp+1);
6
     MPI_Get_address(particle[0].b, disp+2);
7
     base = disp[0];
8
     for (i=0; i < 3; i++) disp[i] = MPI_Aint_diff(disp[i], base);</pre>
9
10
     MPI_Type_create_struct(3, blocklen, disp, type, &Particlestruct);
11
12
     /* Since the compiler may pad the structure, it is best to explicitly
13
        set the extent of the MPI datatype for a structure element using
14
        MPI_Type_create_resized */
15
16
     /* compute extent of the structure */
17
     MPI_Get_address(particle+1, &sizeofentry);
18
     sizeofentry = MPI_Aint_diff(sizeofentry, base);
19
20
     /* build datatype describing structure */
21
     MPI_Type_create_resized(Particlestruct, 0, sizeofentry, &Particletype);
22
23
^{24}
     /* 4.1: send the entire array */
25
26
     MPI_Type_commit(&Particletype);
27
     MPI_Send(particle, 1000, Particletype, dest, tag, comm);
28
29
30
     /* 4.2: send only the entries of type zero particles,
^{31}
             preceded by the number of such entries */
32
33
     MPI_Datatype Zparticles;
                                  /* datatype describing all particles
34
                                     with type zero (needs to be recomputed
35
                                     if types change) */
36
     MPI_Datatype Ztype;
37
38
                   zdisp[1000];
     int
39
                   zblock[1000], j, k;
     int
40
                   zzblock[2] = \{1,1\};
     int
41
                   zzdisp[2];
     MPI_Aint
42
     MPI_Datatype zztype[2];
43
44
     /* compute displacements of type zero particles */
45
     j = 0;
46
     for (i=0; i < 1000; i++)</pre>
47
        if (particle[i].type == 0)
48
           {
```

```
1
        zdisp[j] = i;
                                                                                      \mathbf{2}
        zblock[j] = 1;
                                                                                      3
        j++;
      }
                                                                                      4
                                                                                      5
                                                                                      6
/* create datatype for type zero particles */
MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
                                                                                      7
/* prepend particle count */
                                                                                     9
MPI_Get_address(&j, zzdisp);
                                                                                     10
                                                                                     11
MPI_Get_address(particle, zzdisp+1);
zztype[0] = MPI_INT;
                                                                                     12
zztype[1] = Zparticles;
                                                                                     13
                                                                                     14
MPI_Type_create_struct(2, zzblock, zzdisp, zztype, &Ztype);
                                                                                     15
                                                                                     16
MPI_Type_commit(&Ztype);
                                                                                     17
MPI_Send(MPI_BOTTOM, 1, Ztype, dest, tag, comm);
                                                                                     18
                                                                                     19
/* A probably more efficient way of defining Zparticles */
                                                                                     20
                                                                                     21
/* consecutive particles with index zero are handled as one block */
                                                                                     22
                                                                                     23
j=0;
                                                                                     ^{24}
for (i=0; i < 1000; i++)
                                                                                     25
   if (particle[i].type == 0)
                                                                                     26
      {
         for (k=i+1; (k < 1000)&&(particle[k].type == 0); k++);</pre>
                                                                                     27
         zdisp[j] = i;
                                                                                     28
                                                                                     29
         zblock[j] = k-i;
                                                                                     30
         j++;
                                                                                     31
         i = k;
      }
                                                                                     32
MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
                                                                                     33
                                                                                     34
                                                                                     35
/* 4.3: send the first two coordinates of all entries */
                                                                                     36
                                                                                     37
MPI_Datatype Allpairs;
                              /* datatype for all pairs of coordinates */
                                                                                     38
                                                                                     39
MPI_Type_get_extent(Particletype, &lb, &sizeofentry);
                                                                                     40
                                                                                     41
                                                                                     42
/* sizeofentry can also be computed by subtracting the address
   of particle[0] from the address of particle[1] */
                                                                                     43
                                                                                     44
MPI_Type_create_hvector(1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
                                                                                     45
                                                                                     46
MPI_Type_commit(&Allpairs);
                                                                                     47
MPI_Send(particle[0].d, 1, Allpairs, dest, tag, comm);
                                                                                     48
```

```
1
     /* an alternative solution to 4.3 */
\mathbf{2}
3
     MPI_Datatype Twodouble;
4
\mathbf{5}
     MPI_Type_contiguous(2, MPI_DOUBLE, &Twodouble);
6
7
     MPI_Datatype Onepair;
                               /* datatype for one pair of coordinates, with
8
                                  the extent of one particle entry */
9
10
     MPI_Type_create_resized(Twodouble, 0, sizeofentry, &Onepair );
^{11}
     MPI_Type_commit(&Onepair);
     MPI_Send(particle[0].d, 1000, Onepair, dest, tag, comm);
12
13
14
15
     Example 5.18 The same manipulations as in the previous example, but use absolute
16
     addresses in datatypes.
17
18
     struct Partstruct
19
     {
20
         int
                 type;
21
         double d[6];
22
         char
                 b[7];
23
     };
24
25
     struct Partstruct particle[1000];
26
27
     /* build datatype describing first array entry */
28
29
     MPI_Datatype Particletype;
30
     MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
31
                   block[3] = \{1, 6, 7\};
     int
32
     MPI_Aint
                   disp[3];
33
34
     MPI_Get_address(particle, disp);
35
     MPI_Get_address(particle[0].d, disp+1);
36
     MPI_Get_address(particle[0].b, disp+2);
37
     MPI_Type_create_struct(3, block, disp, type, &Particletype);
38
39
     /* Particletype describes first array entry -- using absolute
40
        addresses */
41
42
     /* 5.1: send the entire array */
43
44
     MPI_Type_commit(&Particletype);
45
     MPI_Send(MPI_BOTTOM, 1000, Particletype, dest, tag, comm);
46
47
48
```

```
1
/* 5.2: send the entries of type zero,
                                                                                        \mathbf{2}
         preceded by the number of such entries */
                                                                                        3
MPI_Datatype Zparticles, Ztype;
                                                                                        4
                                                                                        5
                                                                                        6
              zdisp[1000];
int
int
              zblock[1000], i, j, k;
                                                                                        7
              zzblock[2] = {1,1};
int
                                                                                        9
MPI_Datatype zztype[2];
                                                                                        10
MPI_Aint
              zzdisp[2];
                                                                                        11
j=0;
                                                                                        12
for (i=0; i < 1000; i++)
                                                                                        13
                                                                                        14
    if (particle[i].type == 0)
                                                                                        15
         {
                                                                                        16
             for (k=i+1; (k < 1000)&&(particle[k].type == 0); k++);</pre>
                                                                                        17
             zdisp[j] = i;
                                                                                        18
             zblock[j] = k-i;
                                                                                        19
             j++;
             i = k;
                                                                                        20
                                                                                        21
         }
MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
                                                                                        22
/* Zparticles describe particles with type zero, using
                                                                                        23
                                                                                        ^{24}
   their absolute addresses*/
                                                                                        25
                                                                                        26
/* prepend particle count */
MPI_Get_address(&j, zzdisp);
                                                                                        27
zzdisp[1] = (MPI_Aint)0;
                                                                                        28
                                                                                        29
zztype[0] = MPI_INT;
                                                                                        30
zztype[1] = Zparticles;
MPI_Type_create_struct(2, zzblock, zzdisp, zztype, &Ztype);
                                                                                        31
                                                                                        32
                                                                                        33
MPI_Type_commit(&Ztype);
                                                                                        34
MPI_Send(MPI_BOTTOM, 1, Ztype, dest, tag, comm);
                                                                                        35
                                                                                        36
                                                                                        37
Example 5.19 This example shows how datatypes can be used to handle unions.
                                                                                        38
                                                                                        39
union {
                                                                                        40
   int
            ival;
                                                                                        41
   float
            fval;
                                                                                        42
       } u[1000];
                                                                                        43
                                                                                        44
int
         i, utype;
                                                                                        45
                                                                                        46
/* All entries of u have identical type; variable
                                                                                        47
   utype keeps track of their current type */
                                                                                        48
```

Unofficial Draft for Comment Only

```
1
\mathbf{2}
     MPI_Datatype
                     mpi_utype[2];
3
     MPI_Aint
                     ubase, extent;
4
5
     /* compute an MPI datatype for each possible union type;
6
        assume values are left-aligned in union storage. */
7
8
     MPI_Get_address(u, &ubase);
9
     MPI_Get_address(u+1, &extent);
10
     extent = MPI_Aint_diff(extent, ubase);
11
12
     MPI_Type_create_resized(MPI_INT, 0, extent, &mpi_utype[0]);
13
14
     MPI_Type_create_resized(MPI_FLOAT, 0, extent, &mpi_utype[1]);
15
16
     for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);</pre>
17
18
     /* actual communication */
19
     MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
20
21
     Example 5.20 This example shows how a datatype can be decoded. The routine
22
     printdatatype prints out the elements of the datatype. Note the use of MPI_Type_free for
23
     datatypes that are not predefined.
24
25
     /*
26
       Example of decoding a datatype.
27
28
       Returns 0 if the datatype is predefined, 1 otherwise
29
      */
30
     #include <stdio.h>
^{31}
     #include <stdlib.h>
32
     #include "mpi.h"
33
     int printdatatype(MPI_Datatype datatype)
34
     {
35
         int *array_of_ints;
36
         MPI_Aint *array_of_adds;
37
         MPI_Datatype *array_of_dtypes;
38
         int num_ints, num_adds, num_dtypes, combiner;
39
         int i;
40
41
         MPI_Type_get_envelope(datatype,
42
                                 &num_ints, &num_adds, &num_dtypes, &combiner);
43
         switch (combiner) {
44
         case MPI_COMBINER_NAMED:
45
              printf("Datatype is named:");
46
              /* To print the specific type, we can match against the
47
                 predefined forms. We can NOT use a switch statement here
48
                 We could also use MPI_TYPE_GET_NAME if we prefered to use
```

```
1
       names that the user may have changed.
                                                                                  \mathbf{2}
     */
                                                                                  3
    if
             (datatype == MPI_INT)
                                        printf("MPI_INT\n");
    else if (datatype == MPI_DOUBLE) printf("MPI_DOUBLE\n");
                                                                                  4
    ... else test for other types ...
                                                                                  5
                                                                                  6
    return 0;
    break;
case MPI_COMBINER_STRUCT:
                                                                                  9
case MPI_COMBINER_STRUCT_INTEGER:
                                                                                  10
    printf("Datatype is struct containing");
                                                                                  11
    array_of_ints
                     = (int *)malloc(num_ints * sizeof(int));
    array_of_adds
                                                                                  12
                (MPI_Aint *) malloc(num_adds * sizeof(MPI_Aint));
                                                                                  13
    array_of_dtypes = (MPI_Datatype *)
                                                                                  14
                                                                                  15
        malloc(num_dtypes * sizeof(MPI_Datatype));
                                                                                  16
    MPI_Type_get_contents(datatype, num_ints, num_adds, num_dtypes,
                                                                                  17
                         array_of_ints, array_of_adds, array_of_dtypes);
                                                                                  18
    printf(" %d datatypes:\n", array_of_ints[0]);
                                                                                  19
    for (i=0; i<array_of_ints[0]; i++) {</pre>
        printf("blocklength %d, displacement %ld, type:\n",
                                                                                  20
                                                                                 21
                 array_of_ints[i+1], (long)array_of_adds[i]);
        if (printdatatype(array_of_dtypes[i])) {
                                                                                  22
                                                                                  23
             /* Note that we free the type ONLY if it
                                                                                  24
                is not predefined */
                                                                                  25
             MPI_Type_free(&array_of_dtypes[i]);
                                                                                  26
        }
    }
                                                                                  27
    free(array_of_ints);
                                                                                  28
                                                                                  29
    free(array_of_adds);
    free(array_of_dtypes);
                                                                                  30
                                                                                  31
    break;
                                                                                  32
    . . .
        other combiner values ...
                                                                                  33
default:
                                                                                  34
    printf("Unrecognized combiner type\n");
                                                                                  35
                                                                                  36
return 1;
                                                                                  37
                                                                                  38
                                                                                  39
```

5.2 Pack and Unpack

}

Some existing communication libraries provide pack/unpack functions for sending noncontiguous data. In these, the user explicitly packs data into a contiguous buffer before sending it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are described in Section 5.1, allow one, in most cases, to avoid explicit packing and unpacking. The user specifies the layout of the data to be sent or received, and the communication library directly accesses a noncontiguous buffer. The pack/unpack routines are provided for compatibility with previous libraries. Also, they provide some functionality that is not

Unofficial Draft for Comment Only

40 41

42

43

44

45

46

47

```
1
      otherwise available in MPI. For instance, a message can be received in several parts, where
\mathbf{2}
      the receive operation done on a later part may depend on the content of a former part.
3
      Another use is that outgoing messages may be explicitly buffered in user supplied space,
4
      thus overriding the system buffering policy. Finally, the availability of pack and unpack
5
      operations facilitates the development of additional communication libraries layered on top
6
      of MPI.
7
8
      MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)
9
10
                 inbuf
       IN
                                              input buffer start (choice)
11
       IN
                                               number of input data items (non-negative integer)
                 incount
12
                                               datatype of each input data item (handle)
       IN
                 datatype
13
14
       OUT
                 outbuf
                                              output buffer start (choice)
15
       IN
                 outsize
                                              output buffer size, in bytes (non-negative integer)
16
                                              current position in buffer, in bytes (integer)
       INOUT
                 position
17
18
       IN
                                               communicator for packed message (handle)
                 comm
19
20
      C binding
21
      int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype,
22
                     void *outbuf, int outsize, int *position, MPI_Comm comm)
23
     Fortran 2008 binding
^{24}
     MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
25
          TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
26
          INTEGER, INTENT(IN) :: incount, outsize
27
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
          TYPE(*), DIMENSION(..) :: outbuf
29
          INTEGER, INTENT(INOUT) :: position
30
          TYPE(MPI_Comm), INTENT(IN) :: comm
^{31}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
      Fortran binding
34
      MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
35
          <type> INBUF(*), OUTBUF(*)
36
          INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
37
          Packs the message in the send buffer specified by inbuf, incount, datatype into the buffer
38
      space specified by outbuf and outsize. The input buffer can be any communication buffer
39
      allowed in MPI_SEND. The output buffer is a contiguous storage area containing outsize
40
      bytes, starting at the address outbuf (length is counted in bytes, not elements, as if it were
41
      a communication buffer for a message of type MPI_PACKED).
42
          The input value of position is the first location in the output buffer to be used for
43
      packing. position is incremented by the size of the packed message, and the output value
44
      of position is the first location in the output buffer following the locations occupied by the
45
```

47 48 for sending the packed message.

packed message. The comm argument is the communicator that will be subsequently used

N	/IPI_UNPA	CK(inbut, insize, position, outb	ut, outcount, datatype, comm)	
	IN	inbuf	input buffer start (choice)	2
	IN	insize	size of input buffer, in bytes (non-negative integer)	4
	INOUT	position	current position in bytes (integer)	Ę
	OUT	outbuf	output buffer start (choice)	6
	IN	outcount	number of items to be unpacked (integer)	8
	IN	datatype	datatype of each output data item (handle)	9
	IN	comm	communicator for packed message (handle)	1

MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)

C binding

Fortran 2008 binding

MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
ierror)
TYPE(*), DIMENSION(), INTENT(IN) :: inbuf
INTEGER, INTENT(IN) :: insize, outcount
INTEGER, INTENT(INOUT) :: position
TYPE(*), DIMENSION() :: outbuf
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
IERROR)

<type> INBUF(*), OUTBUF(*)

INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR

Unpacks a message into the receive buffer specified by outbuf, outcount, datatype from the buffer space specified by inbuf and insize. The output buffer can be any communication buffer allowed in MPI_RECV. The input buffer is a contiguous storage area containing insize bytes, starting at address inbuf. The input value of position is the first location in the input buffer occupied by the packed message. position is incremented by the size of the packed message, so that the output value of position is the first location in the input buffer after the locations occupied by the message that was unpacked. comm is the communicator used to receive the packed message.

41 Advice to users. Note the difference between MPI_RECV and MPI_UNPACK: in 42MPI_RECV, the count argument specifies the maximum number of items that can be received. The actual number of items received is determined by the length of 4344the incoming message. In MPI_UNPACK, the count argument specifies the actual number of items that are unpacked; the "size" of the corresponding message is the 4546increment in **position**. The reason for this change is that the "incoming message size" 47is not predetermined since the user decides how much to unpack; nor is it easy to 48 determine the "message size" from the number of items to be unpacked. In fact, in a

Unofficial Draft for Comment Only

13

14

15 16

30

31 32

33

34

35

36

37

38

39

heterogeneous system, this number may not be determined a priori. (End of advice to users.)

To understand the behavior of pack and unpack, it is convenient to think of the data part of a message as being the sequence obtained by concatenating the successive values sent in that message. The pack operation stores this sequence in the buffer space, as if sending the message to that buffer. The unpack operation retrieves this sequence from buffer space, as if receiving a message from that buffer. (It is helpful to think of internal Fortran files or sscanf in C, for a similar function.)

Several messages can be successively packed into one **packing unit**. This is effected by several successive **related** calls to MPI_PACK, where the first call provides **position** = 0, and each successive call inputs the value of **position** that was output by the previous call, and the same values for **outbuf**, **outcount** and **comm**. This packing unit now contains the equivalent information that would have been stored in a message by one send call with a send buffer that is the "concatenation" of the individual send buffers.

A packing unit can be sent using type MPI_PACKED. Any point to point or collective communication function can be used to move the sequence of bytes that forms the packing unit from one process to another. This packing unit can now be received using any receive operation, with any datatype: the type matching rules are relaxed for messages sent with type MPI_PACKED.

A message sent with any type (including MPI_PACKED) can be received using the type MPI_PACKED. Such a message can then be unpacked by calls to MPI_UNPACK.

A packing unit (or a message created by a regular, "typed" send) can be unpacked into
 several successive messages. This is effected by several successive related calls to

MPI_UNPACK, where the first call provides position = 0, and each successive call inputs the
 value of position that was output by the previous call, and the same values for inbuf, insize
 and comm.

The concatenation of two packing units is not necessarily a packing unit; nor is a substring of a packing unit necessarily a packing unit. Thus, one cannot concatenate two packing units and then unpack the result as one packing unit; nor can one unpack a substring of a packing unit as a separate packing unit. Each packing unit, that was created by a related sequence of pack calls, or by a regular send, must be unpacked as a unit, by a sequence of related unpack calls.

Rationale. The restriction on "atomic" packing and unpacking of packing units allows the implementation to add at the head of packing units additional information, such as a description of the sender architecture (to be used for type conversion, in a heterogeneous environment) (*End of rationale.*)

The following call allows the user to find out how much space is needed to pack a message and, thus, manage space allocation for buffers.

 $41 \\ 42$

34

35

36

37

38 39

40

- 43
- 44
- 45
- 46
- 47
- 48

1

2

MPI_P/	ACK_SIZE(incount, datatype, com	m, size)	1
IN	incount	count argument to packing call (non-negative integer)	2
IN	datatype	datatype argument to packing call (handle)	$\frac{3}{4}$
IN	comm	communicator argument to packing call (handle)	5
OUT	size	upper bound on size of packed message, in bytes	6
001	5120	(non-negative integer)	7
			8
C bind	ling		9
int MP	I_Pack_size(int incount, MPI	I_Datatype datatype, MPI_Comm comm,	10 11
	int *size)		12
Fortra	n 2008 binding		13
	ck_size(incount, datatype, o	comm, size, ierror)	14
IN	TEGER, INTENT(IN) :: incount	t i i i i i i i i i i i i i i i i i i i	15
	PE(MPI_Datatype), INTENT(IN)		16
	PE(MPI_Comm), INTENT(IN) ::	comm	17
	TEGER, INTENT(OUT) :: size TEGER, OPTIONAL, INTENT(OUT)		18 19
TN	IEGER, OFIIONAL, INIENI(001)		20
	n binding		21
	CK_SIZE(INCOUNT, DATATYPE, (22
IN	TEGER INCOUNT, DATATYPE, CON	MM, SIZE, IERROR	23
	•	latatype, comm, size) returns in size an upper bound	24
	-	ed by a call to MPI_PACK(inbuf, incount, datatype,	25
		e packed size of the datatype cannot be expressed	26
by the	size parameter, then MPI_PACK_	SIZE sets the value of size to MPI_UNDEFINED.	27 28
		oper bound, rather than an exact bound, since the	29
		back the message may depend on the context (e.g.,	30
fi	rst message packed in a packing	mit may take more space). (End of rationale.)	31
_		24.21	32
Exam	ble 5.21 An example using MPI	_PACK.	33
int	<pre>position, i, j, a[2];</pre>		34
char	buff[1000];		35 36
MDT Co	mm ronk (MDI COMM MODID fmin		37
	mm_rank(MPI_COMM_WORLD, &my) rank == 0)	lalik),	38
11 (my {			39
-	SENDER CODE */		40
			41
ро	sition = 0;		42
		, 1000, &position, MPI_COMM_WORLD);	43
	-	, 1000, &position, MPI_COMM_WORLD);	44 45
MP }	1_Send(DUII, position, MP1_1	PACKED, 1, 0, MPI_COMM_WORLD);	45 46
-	/* RECEIVER CODE */		47
		<pre>MPI_COMM_WORLD, MPI_STATUS_IGNORE);</pre>	48

Unofficial Draft for Comment Only

```
1
     Example 5.22 An elaborate example.
\mathbf{2}
     int
           position, i;
3
     float a[1000];
4
     char buff[1000];
5
6
     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
7
     if (myrank == 0)
8
     {
9
         /* SENDER CODE */
10
11
         int len[2];
12
         MPI_Aint disp[2];
13
         MPI_Datatype type[2], newtype;
14
15
         /* build datatype for i followed by a[0]...a[i-1]
16
17
         len[0] = 1;
18
         len[1] = i;
19
         MPI_Get_address(&i, disp);
20
         MPI_Get_address(a, disp+1);
21
         type[0] = MPI_INT;
22
         type[1] = MPI_FLOAT;
23
         MPI_Type_create_struct(2, len, disp, type, &newtype);
24
         MPI_Type_commit(&newtype);
25
26
         /* Pack i followed by a[0]...a[i-1]*/
27
28
         position = 0;
29
         MPI_Pack(MPI_BOTTOM, 1, newtype, buff, 1000, &position, MPI_COMM_WORLD);
30
31
         /* Send */
32
33
         MPI_Send(buff, position, MPI_PACKED, 1, 0,
34
                   MPI_COMM_WORLD);
35
36
     /* ****
37
        One can replace the last three lines with
38
        MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
39
        **** */
40
     }
41
     else if (myrank == 1)
42
     {
43
         /* RECEIVER CODE */
44
45
         MPI_Status status;
46
47
         /* Receive */
48
```

```
1
                                                                                       \mathbf{2}
    MPI_Recv(buff, 1000, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);
                                                                                       3
    /* Unpack i */
                                                                                       \mathbf{4}
                                                                                       5
                                                                                       6
    position = 0;
    MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);
    /* Unpack a[0]...a[i-1] */
                                                                                       9
                                                                                      10
    MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
}
                                                                                      11
                                                                                      12
                                                                                      13
Example 5.23 Each process sends a count, followed by count characters to the root; the
                                                                                      14
root concatenates all characters into one string.
                                                                                      15
                                                                                      16
int count, gsize, counts[64], totalcount, k1, k2, k,
                                                                                      17
     displs[64], position, concat_pos;
                                                                                      18
char chr[100], *lbuf, *rbuf, *cbuf;
                                                                                      19
MPI_Comm_size(comm, &gsize);
                                                                                      20
                                                                                      21
MPI_Comm_rank(comm, &myrank);
                                                                                      22
                                                                                      23
      /* allocate local pack buffer */
                                                                                      24
MPI_Pack_size(1, MPI_INT, comm, &k1);
                                                                                      25
MPI_Pack_size(count, MPI_CHAR, comm, &k2);
                                                                                      26
k = k1 + k2;
lbuf = (char *)malloc(k);
                                                                                      27
                                                                                      28
                                                                                      29
      /* pack count, followed by count characters */
                                                                                      30
position = 0;
                                                                                      31
MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
                                                                                      32
MPI_Pack(chr, count, MPI_CHAR, lbuf, k, &position, comm);
                                                                                      33
                                                                                      34
if (myrank != root) {
    /* gather at root sizes of all packed messages */
                                                                                      35
                                                                                      36
    MPI_Gather(&position, 1, MPI_INT, NULL, 0,
                MPI_DATATYPE_NULL, root, comm);
                                                                                      37
                                                                                      38
                                                                                      39
    /* gather at root packed messages */
                                                                                      40
    MPI_Gatherv(lbuf, position, MPI_PACKED, NULL,
                                                                                      41
                 NULL, NULL, MPI_DATATYPE_NULL, root, comm);
                                                                                      42
            /* root code */
} else {
                                                                                      43
                                                                                      44
    /* gather sizes of all packed messages */
    MPI_Gather(&position, 1, MPI_INT, counts, 1,
                                                                                      45
                MPI_INT, root, comm);
                                                                                      46
                                                                                      47
                                                                                      48
    /* gather all packed messages */
```

```
1
          displs[0] = 0;
2
          for (i=1; i < gsize; i++)
3
              displs[i] = displs[i-1] + counts[i-1];
4
          totalcount = displs[gsize-1] + counts[gsize-1];
5
          rbuf = (char *)malloc(totalcount);
6
          cbuf = (char *)malloc(totalcount);
7
          MPI_Gatherv(lbuf, position, MPI_PACKED, rbuf,
8
                       counts, displs, MPI_PACKED, root, comm);
9
10
          /* unpack all messages and concatenate strings */
11
          concat_pos = 0;
12
          for (i=0; i < gsize; i++) {</pre>
13
              position = 0;
14
              MPI_Unpack(rbuf+displs[i], totalcount-displs[i],
15
                           &position, &count, 1, MPI_INT, comm);
16
              MPI_Unpack(rbuf+displs[i], totalcount-displs[i],
17
                           &position, cbuf+concat_pos, count, MPI_CHAR, comm);
18
              concat_pos += count;
19
          }
20
          cbuf[concat_pos] = '\0';
21
     }
22
23
            Canonical MPI_PACK and MPI_UNPACK
     5.3
24
25
     These functions read/write data to/from the buffer in the "external32" data format specified
26
     in Section 14.5.2, and calculate the size needed for packing. Their first arguments specify
27
     the data format, for future extensibility, but currently the only valid value of the datarep
28
     argument is "external32."
29
30
           Advice to users. These functions could be used, for example, to send typed data in a
31
           portable format from one MPI implementation to another. (End of advice to users.)
32
          The buffer will contain exactly the packed data, without headers. MPI_BYTE should
33
     be used to send and receive data that is packed using MPI_PACK_EXTERNAL.
34
35
           Rationale. MPI_PACK_EXTERNAL specifies that there is no header on the message
36
           and further specifies the exact format of the data. Since MPI_PACK may (and is
37
           allowed to) use a header, the datatype MPI_PACKED cannot be used for data packed
38
           with MPI_PACK_EXTERNAL. (End of rationale.)
39
40
41
42
43
44
45
46
47
48
```

MPI_PACK_EXTERNAL(datarep, inbuf, incount, datatype, outbuf, outsize, position)			
IN	datarep	data representation (string)	
IN	inbuf	input buffer start (choice)	
IN	incount	number of input data items (integer)	
IN	datatype	datatype of each input data item (handle)	
OUT	outbuf	output buffer start (choice)	
IN	outsize	output buffer size, in bytes (integer)	
INOUT	position	current position in buffer, in bytes (integer)	

C binding

Fortran 2008 binding

MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,
position, ierror)
CHARACTER(LEN=*), INTENT(IN) :: datarep
TYPE(*), DIMENSION(), INTENT(IN) :: inbuf
INTEGER, INTENT(IN) :: incount
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(*), DIMENSION() :: outbuf
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,	
POSITION, IERROR)	
CHARACTER*(*) DATAREP	
<type> INBUF(*), OUTBUF(*)</type>	
INTEGER INCOUNT, DATATYPE, IERROR	
INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION	

```
1
     MPI_UNPACK_EXTERNAL(datarep, inbuf, insize, position, outbuf, outcount, datatype)
\mathbf{2}
       IN
                 datarep
                                             data representation (string)
3
                 inbuf
       IN
                                             input buffer start (choice)
4
5
       IN
                 insize
                                             input buffer size, in bytes (integer)
6
       INOUT
                 position
                                             current position in buffer, in bytes (integer)
7
                 outbuf
       OUT
                                             output buffer start (choice)
8
9
       IN
                                             number of output data items (integer)
                 outcount
10
       IN
                 datatype
                                             datatype of output data item (handle)
11
12
     C binding
13
     int MPI_Unpack_external(const char datarep[], const void *inbuf,
14
                    MPI_Aint insize, MPI_Aint *position, void *outbuf,
15
                     int outcount, MPI_Datatype datatype)
16
17
     Fortran 2008 binding
18
     MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
19
                    datatype, ierror)
20
          CHARACTER(LEN=*), INTENT(IN) :: datarep
21
          TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
22
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
23
24
          TYPE(*), DIMENSION(..) :: outbuf
25
          INTEGER, INTENT(IN) :: outcount
26
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     Fortran binding
29
     MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
30
                    DATATYPE, IERROR)
31
          CHARACTER*(*) DATAREP
32
          <type> INBUF(*), OUTBUF(*)
33
          INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
34
          INTEGER OUTCOUNT, DATATYPE, IERROR
35
36
37
     MPI_PACK_EXTERNAL_SIZE(datarep, incount, datatype, size)
38
39
       IN
                 datarep
                                             data representation (string)
40
       IN
                 incount
                                             number of input data items (integer)
41
42
       IN
                 datatype
                                             datatype of each input data item (handle)
43
       OUT
                 size
                                             output buffer size, in bytes (integer)
44
45
     C binding
46
     int MPI_Pack_external_size(const char datarep[], int incount,
47
                    MPI_Datatype datatype, MPI_Aint *size)
48
```

Fortran 2008 binding MPI_Pack_external_size(datarep, incount, datatype, size, ierror) CHARACTER(LEN=*), INTENT(IN) :: datarep INTEGER, INTENT(IN) :: incount TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR) CHARACTER*(*) DATAREP INTEGER INCOUNT, DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE



Chapter 6

Collective Communication

6.1 Introduction and Overview

Collective communication is defined as communication that involves a group or groups of processes. The functions of this type provided by MPI are the following:

- MPI_BARRIER, MPI_IBARRIER: Barrier synchronization across all members of a group (Section 6.3 and Section 6.12.1).
- MPI_BCAST, MPI_IBCAST: Broadcast from one member to all members of a group (Section 6.4 and Section 6.12.2). This is shown as "broadcast" in Figure 6.1.
- MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV: Gather data from all members of a group to one member (Section 6.5 and Section 6.12.3). This is shown as "gather" in Figure 6.1.
- MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV: Scatter data from one member to all members of a group (Section 6.6 and Section 6.12.4). This is shown as "scatter" in Figure 6.1.
- MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV: A variation on Gather where all members of a group receive the result (Section 6.7 and Section 6.12.5). This is shown as "allgather" in Figure 6.1.
- MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW, MPI_IALLTOALLW: Scatter/Gather data from all members to all members of a group (also called complete exchange) (Section 6.8 and Section 6.12.6). This is shown as "complete exchange" in Figure 6.1.
- MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE, MPI_IREDUCE: Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group (Section 6.9.6 and Section 6.12.8) and a variation where the result is returned to only one member (Section 6.9 and Section 6.12.7).
- MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER: A combined reduction and scatter operation (Section 6.10, Section 6.12.9, and Section 6.12.10).

4

34

35

36

37

38

• MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN: Scan across all members of a group (also called prefix) (Section 6.11, Section 6.11.2, Section 6.12.11, and Section 6.12.12).

One of the key arguments in a call to a collective routine is a communicator that 5defines the group or groups of participating processes and provides a context for the oper-6 ation. This is discussed further in Section 6.2. The syntax and semantics of the collective 7 operations are defined to be consistent with the syntax and semantics of the point-to-point 8 operations. Thus, general datatypes are allowed and must match between sending and re-9 ceiving processes as specified in Chapter 5. Several collective routines such as broadcast 10 and gather have a single originating or receiving process. Such a process is called the *root*. 11 Some arguments in the collective functions are specified as "significant only at root," and 12are ignored for all participants except the root. The reader is referred to Chapter 5 for 13 information concerning communication buffers, general datatypes and type matching rules, 14and to Chapter 7 for information on how to define groups and create communicators. 15

The type-matching conditions for the collective operations are more strict than the corresponding conditions between sender and receiver in point-to-point. Namely, for collective operations, the amount of data sent must exactly match the amount of data specified by the receiver. Different type maps (the layout in memory, see Section 5.1) between sender and receiver are still allowed.

Collective operations can (but are not required to) complete as soon as the caller's 21participation in the collective communication is finished. A blocking operation is complete 22 as soon as the call returns. A nonblocking (immediate) call requires a separate completion 23call (cf. Section 3.7). The completion of a collective operation indicates that the caller is free 24to modify locations in the communication buffer. It does not indicate that other processes 25in the group have completed or even started the operation (unless otherwise implied by the 26description of the operation). Thus, a collective communication operation may, or may not, 27have the effect of synchronizing all participating MPI processes. 28

Collective communication calls may use the same communicators as point-to-point communication; MPI guarantees that messages generated on behalf of collective communication calls will not be confused with messages generated by point-to-point communication. The collective operations do not have a message tag argument. A more detailed discussion of correct use of collective routines is found in Section 6.14.

Rationale. The equal-data restriction (on type matching) was made so as to avoid the complexity of providing a facility analogous to the status argument of MPI_RECV for discovering the amount of data sent. Some of the collective routines would require an array of status values.

The statements about synchronization are made so as to allow a variety of implementations of the collective functions.

(End of rationale.)

Advice to users. It is dangerous to rely on synchronization side-effects of the col lective operations for program correctness. For example, even though a particular
 implementation may provide a broadcast routine with a side-effect of synchroniza tion, the standard does not require this, and a program that relies on this will not be
 portable.

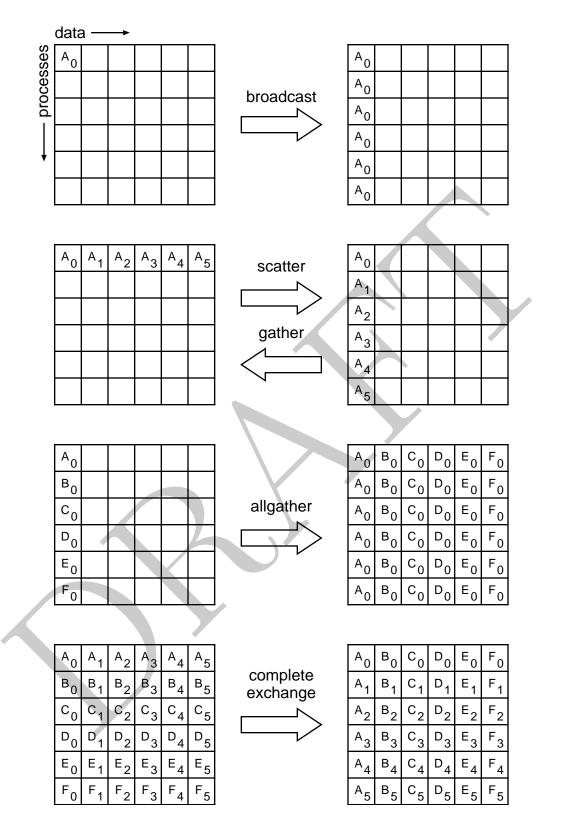


Figure 6.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0 , but after the broadcast all processes contain it.

Unofficial Draft for Comment Only

 31

 $44 \\ 45$

On the other hand, a correct, portable program must allow for the fact that a collective call *may* be synchronizing. Though one cannot rely on any synchronization side-effect, one must program so as to allow it. These issues are discussed further in Section 6.14. (*End of advice to users.*)

Advice to implementors. While vendors may write optimized collective routines matched to their architectures, a complete library of the collective communication routines can be written entirely using the MPI point-to-point communication functions and a few auxiliary functions. If implementing on top of point-to-point, a hidden, special communicator might be created for the collective operation so as to avoid interference with any on-going point-to-point communication at the time of the collective call. This is discussed further in Section 6.14. (End of advice to implementors.)

¹³ Many of the descriptions of the collective routines provide illustrations in terms of ¹⁴ blocking MPI point-to-point routines. These are intended solely to indicate what data is ¹⁵ sent or received by what process. Many of these examples are *not* correct MPI programs; ¹⁶ for purposes of simplicity, they often assume infinite buffering.

17 18

19

6.2 Communicator Argument

The key concept of the collective functions is to have a group or groups of participating processes. The routines do not have group identifiers as explicit arguments. Instead, there is a communicator argument. Groups and communicators are discussed in full detail in Chapter 7. For the purposes of this chapter, it is sufficient to know that there are two types of communicators: *intra-communicators* and *inter-communicators*. An intracommunicator can be thought of as an identifier for a single group of processes linked with a context. An intercommunicator identifies two distinct groups of processes linked with a context.

²⁸ 6.2.1 Specifics for Intracommunicator Collective Operations

All processes in the group identified by the intracommunicator must call the collective routine.

In many cases, collective communication can occur "in place" for intracommunicators, with the output buffer being identical to the input buffer. This is specified by providing a special argument value, MPI_IN_PLACE, instead of the send buffer or the receive buffer argument, depending on the operation performed.

36 The "in place" operations are provided to reduce unnecessary memory Rationale. 37 motion by both the MPI implementation and by the user. Note that while the simple 38 check of testing whether the send and receive buffers have the same address will 39 work for some cases (e.g., MPI_ALLREDUCE), they are inadequate in others (e.g., 40 MPI_GATHER, with root not equal to zero). Further, Fortran explicitly prohibits 41 aliasing of arguments; the approach of using a special value to denote "in place" 42operation eliminates that difficulty. (End of rationale.) 43

Advice to users. By allowing the "in place" option, the receive buffer in many of the
 collective calls becomes a send-and-receive buffer. For this reason, a Fortran binding
 that includes INTENT must mark these as INOUT, not OUT.

⁴⁷ Note that MPI_IN_PLACE is a special kind of value; it has the same restrictions on its ⁴⁸ use that MPI_BOTTOM has. (*End of advice to users.*)

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11

6.2.2 Applying Collective Operations to Intercommunicators
To understand how collective operations apply to intercommunicators, we can view most MPI intracommunicator collective operations as fitting one of the following categories (see, for instance, [63]):
All-To-All All processes contribute to the result. All processes receive the result.
 MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV
• MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW
 MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER 1
• MPI_BARRIER, MPI_IBARRIER
All-To-One All processes contribute to the result. One process receives the result.
MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV MPI_REDUCE, MPI_IREDUCE
One-To-All One process contributes to the result. All processes receive the result.
MPI_BCAST, MPI_IBCAST MPI_SCATTER, MPI_SCATTERV, MPI_ISCATTERV 2
Other Collective operations that do not fit into one of the above categories.
MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN
The data movement patterns of MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, and MPI_IEXSCAN do not fit this taxonomy. The application of collective communication to intercommunicators is best described in terms of two groups. For example, an all-to-all MPI_ALLGATHER operation can be described as collecting data from all members of one group with the result appearing in all members of the other group (see Figure 6.2). As another example, a one-to-all MPI_BCAST operation sends data from one member of one group to all members of the other group. Collective computation operations such as MPI_REDUCE_SCATTER have a similar interpretation (see Figure 6.3). For intracommunicators, these two groups are the same. For intercommunicators, these two groups are distinct. For the all-to-all operations, each such operation is described in two phases, so that it has a symmetric, full-duplex behavior.
The following collective operations also apply to intercommunicators: 4
• MPI_BARRIER, MPI_IBARRIER 4
• MPI_BCAST, MPI_IBCAST 4
• MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV, 4
• MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV, 4

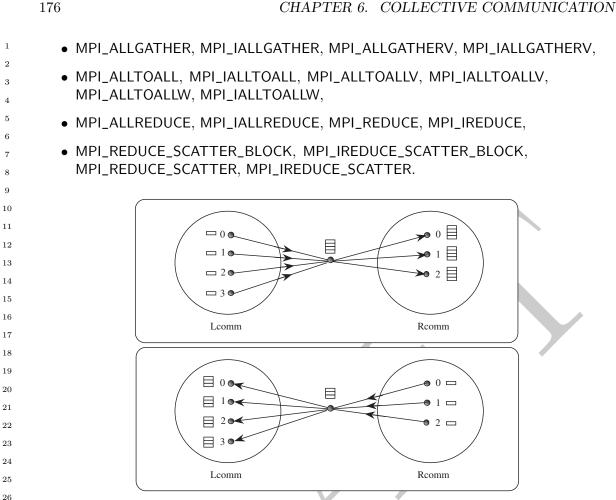


Figure 6.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

29 30 31

32

27

28

Specifics for Intercommunicator Collective Operations 6.2.3

All processes in both groups identified by the intercommunicator must call the collective 33 routine.

34 Note that the "in place" option for intracommunicators does not apply to intercom-35 municators since in the intercommunicator case there is no communication from a process 36 to itself. 37

For intercommunicator collective communication, if the operation is in the All-To-One 38 or One-To-All categories, then the transfer is unidirectional. The direction of the transfer is 39 indicated by a special value of the root argument. In this case, for the group containing the 40 root process, all processes in the group must call the routine using a special argument for 41 the root. For this, the root process uses the special root value MPI_ROOT; all other processes 42in the same group as the root use MPI_PROC_NULL. All processes in the other group (the 43 group that is the remote group relative to the root process) must call the collective routine 44 and provide the rank of the root. If the operation is in the All-To-All category, then the 45 transfer is bidirectional. 46

47Rationale. Operations in the All-To-One and One-To-All categories are unidirectional 48 by nature, and there is a clear way of specifying direction. Operations in the All-To-All

Unofficial Draft for Comment Only

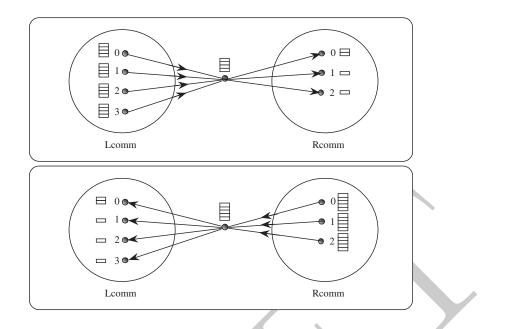


Figure 6.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

category will often occur as part of an exchange, where it makes sense to communicate in both directions at once. (*End of rationale.*)

6.3 Barrier Synchronization

MPI_BARRIER(comm)	29
	30
IN comm com	municator (handle) 31
	32
C binding	33
<pre>int MPI_Barrier(MPI_Comm comm)</pre>	34
	35
Fortran 2008 binding	36
MPI_Barrier(comm, ierror)	37
TYPE(MPI_Comm), INTENT(IN) :: comm	38
INTEGER, OPTIONAL, INTENT(OUT) ::	ierror 39
Fortran binding	40
MPI_BARRIER(COMM, IERROR)	41
INTEGER COMM, IERROR	42
	43
If comm is an intracommunicator, MPI_B	$ARRIER \text{ blocks the caller until all group mem-}_{44}$
bers have called it. The call returns at any pro-	cess only after all group members have entered $_{45}$

the call. If comm is an intercommunicator, MPI_BARRIER involves two groups. The call returns at processes in one group (group A) of the intercommunicator only after all members of the

Unofficial Draft for Comment Only

 $\frac{24}{25}$

 48

```
1
     other group (group B) have entered the call (and vice versa). A process may return from
\mathbf{2}
      the call before all processes in its own group have entered the call.
3
4
            Broadcast
      6.4
5
6
7
8
      MPI_BCAST(buffer, count, datatype, root, comm)
9
       INOUT
                 buffer
                                               starting address of buffer (choice)
10
       IN
                                              number of entries in buffer (non-negative integer)
                 count
11
12
       IN
                 datatype
                                               data type of buffer (handle)
13
       IN
                  root
                                              rank of broadcast root (integer)
14
       IN
                 comm
                                               communicator (handle)
15
16
17
      C binding
      int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
18
19
                     MPI_Comm comm)
20
      Fortran 2008 binding
21
     MPI_Bcast(buffer, count, datatype, root, comm, ierror)
22
          TYPE(*), DIMENSION(..) :: buffer
23
          INTEGER, INTENT(IN) :: count, root
24
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
          TYPE(MPI_Comm), INTENT(IN) :: comm
26
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     Fortran binding
28
      MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
29
30
          <type> BUFFER(*)
^{31}
          INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
32
          If comm is an intracommunicator, MPI_BCAST broadcasts a message from the process
33
      with rank root to all processes of the group, itself included. It is called by all members of
34
      the group using the same arguments for comm and root. On return, the content of root's
35
     buffer is copied to all other processes.
36
          General, derived datatypes are allowed for datatype. The type signature of count,
37
      datatype on any process must be equal to the type signature of count, datatype at the root.
38
      This implies that the amount of data sent must be equal to the amount received, pairwise
39
      between each process and the root. MPI_BCAST and all other data-movement collective
40
     routines make this restriction. Distinct type maps between sender and receiver are still
41
      allowed.
42
          The "in place" option is not meaningful here.
43
          If comm is an intercommunicator, then the call involves all processes in the intercom-
44
      municator, but with one group (group A) defining the root process. All processes in the
45
      other group (group B) pass the same value in argument root, which is the rank of the root
46
      in group A. The root passes the value MPI_ROOT in root. All other processes in group A
```

CHAPTER 6. COLLECTIVE COMMUNICATION

47

48

pass the value MPI_PROC_NULL in root. Data is broadcast from the root to all processes

in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

6.4.1 Example using MPI_BCAST

The examples in this section use intracommunicators.

Example 6.1 Broadcast 100 ints from process 0 to every process in the group.

```
MPI_Comm comm;
int array[100];
int root=0;
...
MPI_Bcast(array, 100, MPI_INT, root, comm);
```

As in many of our example code fragments, we assume that some of the variables (such as comm in the above) have been assigned appropriate values.

6.5 Gather

			21
MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)			22
IN			23
	starting address of send buller (choice)		24
IN	sendcount	number of elements in send buffer (non-negative	25
		integer)	26
IN	sendtype	data type of send buffer elements (handle)	27
	recvbuf	danse formering heffer (sheire simiferent enhant	28
OUT	recvbui	address of receive buffer (choice, significant only at	29
		root)	30
IN	recvcount	number of elements for any single receive	31
		(non-negative integer, significant only at root)	32
IN	recvtype	data type of recv buffer elements (handle, significant	33
	receipe	only at root)	34
		* ,	35
IN	root	rank of receiving process (integer)	36
IN	comm	communicator (handle)	37
C binding	g		39
	2	, int sendcount, MPI_Datatype sendtype,	40
		ecvcount, MPI_Datatype recvtype, int root,	41
	MPI_Comm comm)		42
_			43
Fortran 2	008 binding		44
MPI_Gathe		dtype, recvbuf, recvcount, recvtype,	45
	root, comm, ierror)		46
	*), DIMENSION(), INTENT		47
INTEG	INTEGER, INTENT(IN) :: sendcount, recvcount, root 4		

Unofficial Draft for Comment Only

1 2

 $\frac{3}{4}$

5

6 7

8 9

10

11

12

13

14 15

16

17 18

1	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
2	TYPE(*), DIMENSION() :: recvbuf
3	TYPE(MPI_Comm), INTENT(IN) :: comm
4	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5	Fortran binding
6	MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
7	ROOT, COMM, IERROR)
8 9	<type> SENDBUF(*), RECVBUF(*)</type>
10	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
11	If comm is an intracommunicator, each process (root process included) sends the con-
12	tents of its send buffer to the root process. The root process receives the messages and stores
13	them in rank order. The outcome is as if each of the n processes in the group (including
14	the root process) had executed a call to
15	
16	MPI_Send(sendbuf, sendcount, sendtype, root ,),
17	
18	and the root had executed n calls to
19	MPI_Recv(recvbuf+i· recvcount· extent(recvtype), recvcount, recvtype, i,),
20	
21 22	where extent(recvtype) is the type extent obtained from a call to MPI_Type_get_extent.
23	An alternative description is that the n messages sent by the processes in the group
24	are concatenated in rank order, and the resulting message is received by the root as if by a
25	call to MPI_RECV(recvbuf, recvcount·n, recvtype,).
26	The receive buffer is ignored for all non-root processes.
27	General, derived datatypes are allowed for both sendtype and recvtype. The type signa-
28	ture of sendcount, sendtype on each process must be equal to the type signature of recvcount,
29	recvtype at the root. This implies that the amount of data sent must be equal to the amount
30	of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.
31	All arguments to the function are significant on process root , while on other processes,
32	only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments
33	root and comm must have identical values on all processes.
34	The specification of counts and types should not cause any location on the root to be
35	written more than once. Such a call is erroneous.
36	Note that the recvcount argument at the root indicates the number of items it receives
37	from each process, not the total number of items it receives.
38	The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as
39	the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and
40 41	the contribution of the root to the gathered vector is assumed to be already in the correct
42	place in the receive buffer.
43	If comm is an intercommunicator, then the call involves all processes in the intercom-
44	municator, but with one group (group A) defining the root process. All processes in the
45	other group (group B) pass the same value in argument root, which is the rank of the root
46	in group A. The root passes the value MPI_ROOT in root. All other processes in group A
47	pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to
48	

the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

	comm)		6
IN	sendbuf	starting address of send buffer (choice)	7
IN	sendcount	number of elements in send buffer (non-negative integer)	8 9 10
IN	sendtype	data type of send buffer elements (handle)	11
OUT	recvbuf	address of receive buffer (choice, significant only at root)	12 13
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)	14 15 16
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)	17 18 19 20 21
IN	recvtype	data type of recv buffer elements (handle, significant only at root)	22 23
IN	root	rank of receiving process (integer)	24 25
IN	comm	communicator (handle)	26 27
C bindi	ng		28

```
C binding
```

```
int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
             void *recvbuf, const int recvcounts[], const int displs[],
             MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Fortran 2008 binding

COMM, IERROR

```
33
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  34
             recvtype, root, comm, ierror)
                                                                                  35
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  36
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
                                                                                  37
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  38
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  39
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  41
                                                                                  42
Fortran binding
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
                                                                                  43
             RECVTYPE, ROOT, COMM, IERROR)
                                                                                  44
    <type> SENDBUF(*), RECVBUF(*)
                                                                                  45
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
                                                                                  46
```

1

 $\mathbf{2}$

3 4 5

29

30

3132

MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count

of data from each process, since recvcounts is now an array. It also allows more flexibility as to where the data is placed on the root, by providing the new argument, displs. If comm is an intracommunicator, the outcome is as if each process, including the root process, sends a message to the root, MPI_Send(sendbuf, sendcount, sendtype, root, ...), and the root executes **n** receives. MPI_Recv(recvbuf+displs[j]· extent(recvtype), recvcounts[j], recvtype, i, ...). The data received from process j is placed into recvbuf of the root process beginning at offset displs[j] elements (in terms of the recvtype). The receive buffer is ignored for all non-root processes. The type signature implied by sendcount, sendtype on process i must be equal to the type signature implied by recvcounts[i], recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed, as illustrated in Example 6.6. All arguments to the function are significant on process root, while on other processes, only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments root and comm must have identical values on all processes. The specification of counts, types, and displacements should not cause any location on the root to be written more than once. Such a call is erroneous. The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer. If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root. Examples using MPI_GATHER, MPI_GATHERV 6.5.1 The examples in this section use intracommunicators. **Example 6.2** Gather 100 ints from every process in group to root. See Figure 6.4. MPI_Comm comm; int gsize, sendarray[100]; int root, *rbuf; . . . MPI_Comm_size(comm, &gsize); rbuf = (int *)malloc(gsize*100*sizeof(int));

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

7 8

9 10

11 12

13

14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

 31

32

33

34

35

36 37

38 39

40

41 42

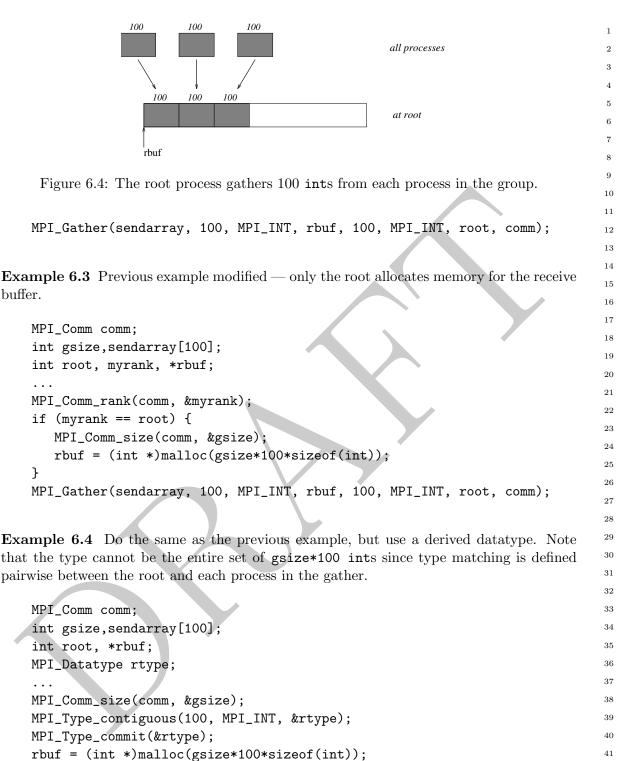
43

44

45

46

47



42 43 44

45

46

47 48

Example 6.5 Now have each process send 100 ints to root, but place each set (of 100)

stride ints apart at receiving end. Use MPI_GATHERV and the displs argument to achieve

MPI_Gather(sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);

this effect. Assume $stride \ge 100$. See Figure 6.5.

```
100
                                  100
                                           100
1
2
                                                                 all processes
3
4
                             100
                                    100
                                           100
5
                                                                 at root
6
7
                                    stride
                           rbuf
8
9
     Figure 6.5: The root process gathers 100 ints from each process in the group, each set is
10
     placed stride ints apart.
11
12
          MPI_Comm comm;
13
          int gsize,sendarray[100];
14
          int root, *rbuf, stride;
15
          int *displs,i,*rcounts;
16
17
          . . .
18
19
          MPI_Comm_size(comm, &gsize);
20
          rbuf = (int *)malloc(gsize*stride*sizeof(int));
21
          displs = (int *)malloc(gsize*sizeof(int));
22
          rcounts = (int *)malloc(gsize*sizeof(int));
23
          for (i=0; i<gsize; ++i) {</pre>
24
               displs[i] = i*stride;
25
               rcounts[i] = 100;
26
          }
27
          MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
28
                       root, comm);
29
30
          Note that the program is erroneous if stride < 100.
^{31}
32
     Example 6.6 Same as Example 6.5 on the receiving side, but send the 100 ints from the
33
     0th column of a 100 \times 150 int array, in C. See Figure 6.6.
34
35
          MPI_Comm comm;
36
          int gsize, sendarray[100][150];
37
          int root, *rbuf, stride;
38
          MPI_Datatype stype;
39
          int *displs,i,*rcounts;
40
41
          . . .
42
          MPI_Comm_size(comm, &gsize);
43
44
          rbuf = (int *)malloc(gsize*stride*sizeof(int));
45
          displs = (int *)malloc(gsize*sizeof(int));
46
          rcounts = (int *)malloc(gsize*sizeof(int));
47
          for (i=0; i<gsize; ++i) {</pre>
48
               displs[i] = i*stride;
```

184

Unofficial Draft for Comment Only

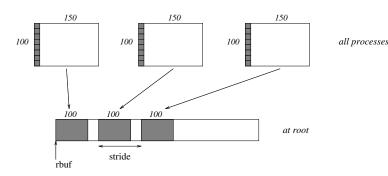


Figure 6.6: The root process gathers column 0 of a 100×150 C array, and each set is placed stride ints apart.

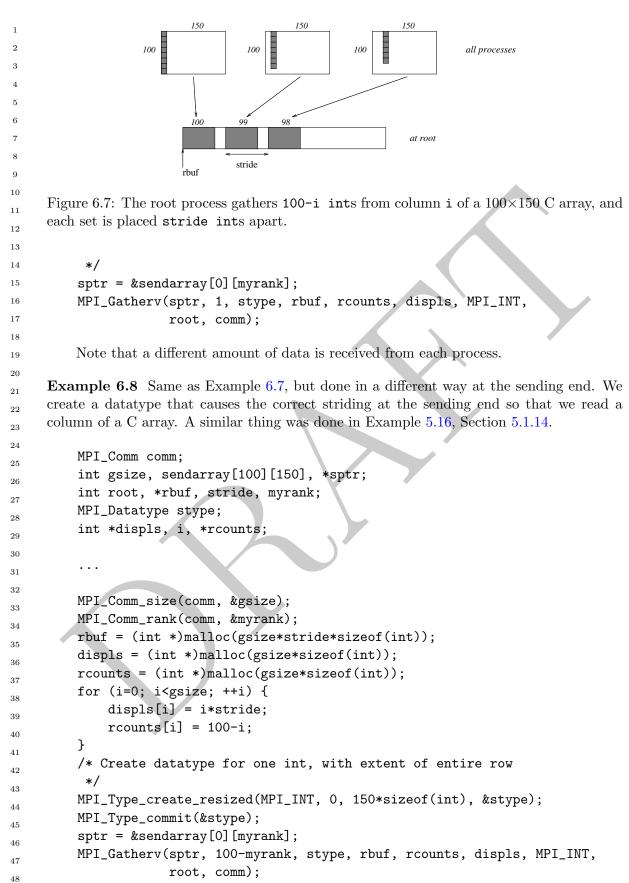
```
rcounts[i] = 100;
}
/* Create datatype for 1 column of array
*/
MPI_Type_vector(100, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
MPI_Gatherv(sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT,
root, comm);
```

Example 6.7 Process i sends (100-i) ints from the i-th column of a 100×150 int array, in C. It is received into a buffer with stride, as in the previous two examples. See Figure 6.7.

```
MPI_Comm comm;
                                                                                 27
int gsize,sendarray[100][150],*sptr;
                                                                                 28
                                                                                 29
int root, *rbuf, stride, myrank;
                                                                                 30
MPI_Datatype stype;
                                                                                 31
int *displs,i,*rcounts;
                                                                                 32
                                                                                 33
                                                                                 34
MPI_Comm_size(comm, &gsize);
                                                                                 35
MPI_Comm_rank(comm, &myrank);
                                                                                 36
                                                                                 37
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
                                                                                 38
                                                                                 39
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {</pre>
                                                                                 40
                                                                                 41
    displs[i] = i*stride;
                                                                                 42
    rcounts[i] = 100-i;
                              /* note change from previous example */
}
                                                                                 43
                                                                                 44
/* Create datatype for the column we are sending
                                                                                 45
 */
                                                                                 46
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
                                                                                 47
MPI_Type_commit(&stype);
                                                                                 48
/* sptr is the address of start of "myrank" column
```

Unofficial Draft for Comment Only

 24



Example 6.9 Same as Example 6.7 at sending side, but at receiving side we make the stride between received blocks vary from block to block. See Figure 6.8.

```
MPI_Comm comm;
int gsize, sendarray[100][150], *sptr;
int root, *rbuf, *stride, myrank, bufsize;
MPI_Datatype stype;
int *displs,i,*rcounts,offset;
. . .
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
stride = (int *)malloc(gsize*sizeof(int));
. . .
/* stride[i] for i = 0 to gsize-1 is set somehow
 */
/* set up displs and rcounts vectors first
 */
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {</pre>
    displs[i] = offset;
    offset += stride[i];
    rcounts[i] = 100-i;
}
/* the required buffer size for rbuf is now easily obtained
 */
bufsize = displs[gsize-1]+rcounts[gsize-1];
rbuf = (int *)malloc(bufsize*sizeof(int));
/* Create datatype for the column we are sending
 */
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
            root, comm);
```

Example 6.10 Process i sends num ints from the i-th column of a 100×150 int array, in C. The complicating factor is that the various values of num are not known to root, so a separate gather must first be run to find these out. The data is placed contiguously at the receiving end.

1

 $\mathbf{2}$

3

4

5

6

7

9 10

11

12

13 14

15 16

17

18 19

20

21

22

23

 24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

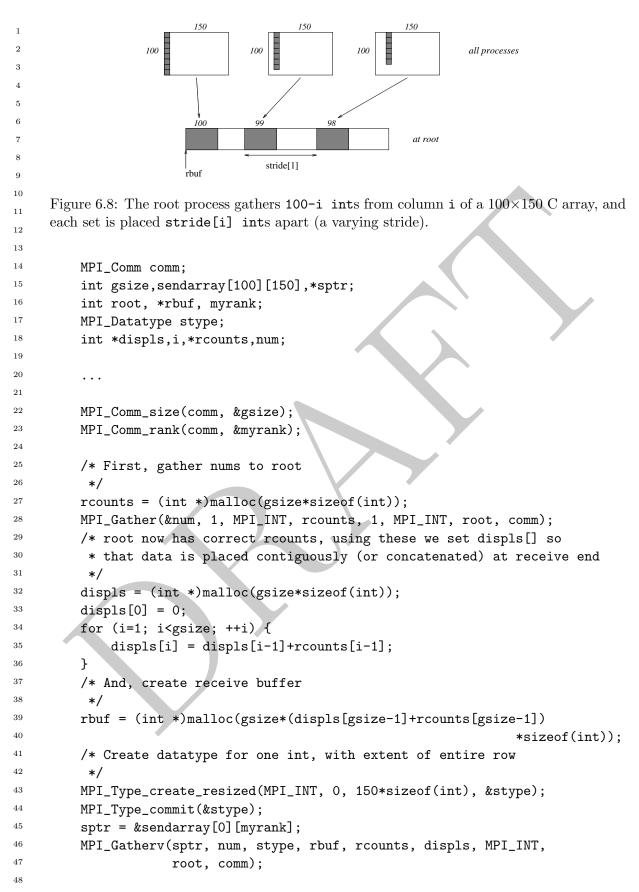
40 41 42

43

44

45

```
CHAPTER 6. COLLECTIVE COMMUNICATION
```



6.6 Scatter

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address of send buffer (choice, significant only at	6
		root)	7
IN	sendcount	number of elements sent to each process	8
		(non-negative integer, significant only at root)	9
IN	sendtype	data type of send buffer elements (handle, significant	10 11
		only at root)	11
OUT	recybuf	address of receive buffer (choice)	13
IN	recvcount	number of elements in receive buffer (non-negative	14
IIN		integer)	15
INI			16
IN	recvtype	data type of receive buffer elements (handle)	17
IN	root	rank of sending process (integer)	18
IN	comm	communicator (handle)	19
			20

C binding

int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Fortran 2008 binding

MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,	27
root, comm, ierror)	28
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf	29
INTEGER, INTENT(IN) :: sendcount, recvcount, root	30
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	31
TYPE(*), DIMENSION() :: recvbuf	32
TYPE(MPI_Comm), INTENT(IN) :: comm	33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
	35

Fortran binding

MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTER is the inverse operation to MPI_GATHER.

If comm is an intracommunicator, the outcome is as if the root executed n send operations,

 $\mathsf{MPI}_\mathsf{Send}(\mathsf{sendbuf}+\mathsf{i}\cdot \mathsf{sendcount}\cdot \mathsf{extent}(\mathsf{sendtype}), \mathsf{sendcount}, \mathsf{sendtype}, \mathsf{i}, ...),$

and each process executed a receive,

5

21

22

23

24

25

26

36

37

38

39

40

41 42

43 44

45 46

1 2

 $\frac{3}{4}$

190

MPI_Recv(recvbuf, recvcount, recvtype, i,...).

An alternative description is that the root sends a message with MPI_Send(sendbuf, sendcount n, sendtype, ...). This message is split into n equal segments, the *i*-th segment is sent to the *i*-th process in the group, and each process receives this message as above.

5 6

17

18

19

20 21

The send buffer is ignored for all non-root processes.

The type signature associated with sendcount, sendtype at the root must be equal to the type signature associated with recvcount, recvtype at all processes (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes,
 only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments
 root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be read more than once.

Rationale. Though not needed, the last restriction is imposed so as to achieve symmetry with MPI_GATHER, where the corresponding restriction (a multiple-write restriction) is necessary. (*End of rationale.*)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain n segments, where n is the group size; the *root*-th segment, which root should "send to itself," is not moved.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

MPI_SCA	`	s, displs, sendtype, recvbuf, recvcount, recvtype, root,	1
	comm)		2 3
IN	sendbuf	address of send buffer (choice, significant only at root)	4
IN	sendcounts	,	5
IIN	senucounts	non-negative integer array (of length group size) specifying the number of elements to send to each	6
		rank (significant only at root)	7 8
IN	displs	integer array (of length group size). Entry i specifies	9
		the displacement (relative to $sendbuf)$ from which to	10
		take the outgoing data to process i (significant only	11
		at root)	12
IN	sendtype	data type of send buffer elements (handle, significant	13
		only at root)	14 15
OUT	recvbuf	address of receive buffer (choice)	16
IN	recvcount	number of elements in receive buffer (non-negative	17
		integer)	18
IN	recvtype	data type of receive buffer elements (handle)	19
IN	root	rank of sending process (integer)	20
			21 22
IN	comm	communicator (handle)	22
C bindir	8		25
<pre>int MPI_Scatterv(const void *sendbuf, const int sendcounts[],</pre>			26
	-	I_Datatype recvtype, int root, MPI_Comm comm)	27
		bababype recouppe, into rece, in r_comm. comm.	28
	2008 binding		29
MPI_Scat		ts, displs, sendtype, recvbuf, recvcount,	30
ירועיד	recvtype, root, c		31
	(*), DIMENSION(), IN	dcounts(*), displs(*), recvcount, root	32
		(IN) :: sendtype, recvtype	33 34
	(*), DIMENSION() :: :	VI VI	34 35
	(MPI_Comm), INTENT(IN)		36
	GER, OPTIONAL, INTENT(37
Fortran	hinding		38
	U	TS DISDIS SENDIVDE RECURILE RECUCOUNT	39
III 1_BOAI	MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)		

<type> SENDBUF(*), RECVBUF(*)

INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

MPI_SCATTERV is the inverse operation to MPI_GATHERV.

MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying 46 count of data to be sent to each process, since sendcounts is now an array. It also allows 47

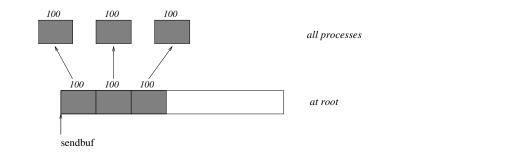
42

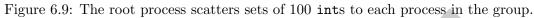
43

44

45

1 more flexibility as to where the data is taken from on the root, by providing an additional $\mathbf{2}$ argument, displs. 3 If comm is an intracommunicator, the outcome is as if the root executed n send oper-4 ations, 5MPI_Send(sendbuf+displs[i] extent(sendtype), sendcounts[i], sendtype, i,...), 6 7 and each process executed a receive, 8 9 MPI_Recv(recvbuf, recvcount, recvtype, i,...). 10 11 The send buffer is ignored for all non-root processes. 12The type signature implied by sendcount[i], sendtype at the root must be equal to the 13 type signature implied by recvcount, recvtype at process i (however, the type maps may be 14different). This implies that the amount of data sent must be equal to the amount of data 15received, pairwise between each process and the root. Distinct type maps between sender 16and receiver are still allowed. 17All arguments to the function are significant on process root, while on other processes, 18 only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments 19root and comm must have identical values on all processes. 20The specification of counts, types, and displacements should not cause any location on 21the root to be read more than once. 22 The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as 23the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and 24 root "sends" no data to itself. The scattered vector is still assumed to contain n segments, 25where n is the group size; the *root*-th segment, which root should "send to itself," is not 26moved. 27If comm is an intercommunicator, then the call involves all processes in the intercom-28municator, but with one group (group A) defining the root process. All processes in the 29 other group (group B) pass the same value in argument root, which is the rank of the root 30 in group A. The root passes the value MPL_ROOT in root. All other processes in group A 31 pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in 32 group B. The receive buffer arguments of the processes in group B must be consistent with 33 the send buffer argument of the root. 34 35 Examples using MPI_SCATTER, MPI_SCATTERV 6.6.1 36 37 The examples in this section use intracommunicators. 38 **Example 6.11** The reverse of Example 6.2. Scatter sets of 100 ints from the root to each 39 process in the group. See Figure 6.9. 4041 MPI_Comm comm; 42int gsize,*sendbuf; 43 int root, rbuf[100]; 44 . . . 45MPI_Comm_size(comm, &gsize); 46 sendbuf = (int *)malloc(gsize*100*sizeof(int)); 47 48 . . .





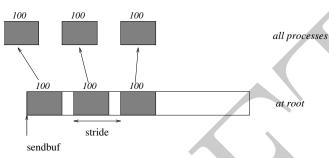


Figure 6.10: The root process scatters sets of 100 ints, moving by stride ints from send to send in the scatter.

MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);

Example 6.12 The reverse of Example 6.5. The root process scatters sets of 100 ints to the other processes, but the sets of 100 are stride ints apart in the sending buffer. Requires use of MPI_SCATTERV. Assume $stride \ge 100$. See Figure 6.10.

```
MPI_Comm comm;
                                                                                30
int gsize,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;
                                                                                34
• • •
                                                                                35
MPI_Comm_size(comm, &gsize);
                                                                                36
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
                                                                                37
. . .
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {</pre>
    displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT,
             root, comm);
```

 24

 31

Example 6.13 The reverse of Example 6.9. We have a varying stride between blocks at
 sending (root) side, at the receiving side we receive into the i-th column of a 100×150 C
 array. See Figure 6.11.

```
4
         MPI_Comm comm;
5
         int gsize,recvarray[100][150],*rptr;
6
         int root, *sendbuf, myrank, *stride;
7
         MPI_Datatype rtype;
8
9
         int i, *displs, *scounts, offset;
10
          . . .
         MPI_Comm_size(comm, &gsize);
11
         MPI_Comm_rank(comm, &myrank);
12
13
         stride = (int *)malloc(gsize*sizeof(int));
14
15
          . . .
16
         /* stride[i] for i = 0 to gsize-1 is set somehow
           * sendbuf comes from elsewhere
17
           */
18
19
          . . .
         displs = (int *)malloc(gsize*sizeof(int));
20
         scounts = (int *)malloc(gsize*sizeof(int));
21
         offset = 0;
22
         for (i=0; i<gsize; ++i) {</pre>
23
^{24}
              displs[i] = offset;
              offset += stride[i];
25
26
              scounts[i] = 100 - i;
         }
27
          /* Create datatype for the column we are receiving
28
           */
29
         MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &rtype);
30
         MPI_Type_commit(&rtype);
31
         rptr = &recvarray[0][myrank];
32
         MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype,
33
34
                        root, comm);
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

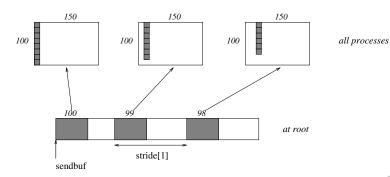


Figure 6.11: The root scatters blocks of 100-i ints into column i of a 100×150 C array. At the sending side, the blocks are stride[i] ints apart.

6.7 Gather-to-all

MPI_ALLO	GATHER(sendbuf, sendcount, s	endtype, recvbuf, recvcount, recvtype, comm)	17 18
IN	sendbuf	starting address of send buffer (choice)	19
IN	sendcount	number of elements in send buffer (non-negative integer)	20 21
IN	sendtype	data type of send buffer elements (handle)	22
OUT	recvbuf		23
001	recybul	address of receive buffer (choice)	24
IN	recvcount	number of elements received from any process	25 26
		(non-negative integer)	20
IN	recvtype	data type of receive buffer elements (handle)	28
IN	comm	communicator (handle)	29
			30
C binding			31
int MPI_	Allgather(const void *sen	dbuf, int sendcount,	32
	MPI_Datatype sendtyp	e, void *recvbuf, int recvcount,	33
	MPI_Datatype recvtyp	e, MPI_Comm comm)	34
Fortran 3	2008 binding		35
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,			36
0	comm, ierror)	,,, _,, _	37
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf			38 39
	GER, INTENT(IN) :: sendco		40
TYPE	(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	40
TYPE	(*), DIMENSION() :: rec	vbuf	42
TYPE	(MPI_Comm), INTENT(IN) ::	comm	43
INTE	GER, OPTIONAL, INTENT(OUT) :: ierror	44
Fortran l	binding		45
	0	SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	46
	COMM, IERROR)	· · · · · · · · · · · · · · · · · · ·	47
<type< td=""><td>e> SENDBUF(*), RECVBUF(*)</td><td></td><td>48</td></type<>	e> SENDBUF(*), RECVBUF(*)		48

 $\mathbf{2}$

1	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
2	
3	MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive
4	the result, instead of just the root. The block of data sent from the j-th process is received
5	by every process and placed in the j-th block of the buffer recvbuf.
6	The type signature associated with sendcount, sendtype, at a process must be equal to
7	the type signature associated with recvcount, recvtype at any other process.
8	If comm is an intracommunicator, the outcome of a call to MPI_ALLGATHER() is as
9	if all processes executed n calls to
10	MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount,
11	recvtype,root,comm)
12	
13	for root = 0,, n-1. The rules for correct usage of MPI_ALLGATHER are easily found
14	from the corresponding rules for MPI_GATHER.
15	The "in place" option for intracommunicators is specified by passing the value
16	MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored.
17	Then the input data of each process is assumed to be in the area where that process would
18 19	receive its own contribution to the receive buffer. If comm is an intercommunicator, then each process of one group (group A) contributes
20	sendcount data items; these data are concatenated and the result is stored at each process
20	in the other group (group B). Conversely the concatenation of the contributions of the
22	processes in group B is stored at each process in group A. The send buffer arguments in
23	group A must be consistent with the receive buffer arguments in group B, and vice versa.
24	
25	Advice to users. The communication pattern of MPI_ALLGATHER executed on an
26	intercommunication domain need not be symmetric. The number of items sent by
27	processes in group A (as specified by the arguments sendcount, sendtype in group A
28	and the arguments recvcount, recvtype in group B), need not equal the number of
29	items sent by processes in group B (as specified by the arguments sendcount, sendtype
30	in group B and the arguments recvcount, recvtype in group A). In particular, one can
31	move data in only one direction by specifying $sendcount = 0$ for the communication
32	in the reverse direction. (End of advice to users.)
33	
34	
35 36	
30 37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, ¹ comm) ²				
IN	sendbuf	starting address of send buffer (choice)	3	
IN	sendcount		4	
IIN	senacount	number of elements in send buffer (non-negative integer)	5 6	
IN	sendtype	data type of send buffer elements (handle)	7	
OUT	recvbuf	address of receive buffer (choice)	8	
			9	
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process	10 11 12	
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	13 14 15	
IN	recvtype	data type of receive buffer elements (handle)	16	
IN	comm	communicator (handle)	17 18	
			19	
C bindin	g		20	
int MPI_	Allgatherv(const void *se	ndbuf, int sendcount,	21	
in i_bacatjpe benacjpe, tera iteribal, tense int iteribalist,			22	
			23	
Fortran 2008 binding			24	
MPI Allgathery(sendbuf, sendcount, sendtype, recybuf, recycounts, displs,			25 26	
	recvtype, comm, ierr		27	
	(*), DIMENSION(), INTEN		28	
		<pre>unt, recvcounts(*), displs(*)</pre>	29	
	(MPI_Datatype), INTENT(IN		30	
	(*), DIMENSION() :: rec		31	
	(MPI_Comm), INTENT(IN) :: GER, OPTIONAL, INTENT(OUT		32	
	JER, OFIIONAL, INTENI(001) ieiioi	33	
Fortran	3		34	
MPI_ALLG		, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,	35	
	RECVTYPE, COMM, IERR	LOR)	36	
01	e> SENDBUF(*), RECVBUF(*)		37	
INTE	JER SENDCOUNT, SENDTYPE, JIERROR	RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	38 39	
			40	
	8	t of as MPI_GATHERV, but where all processes re-	41	
ceive the result, instead of just the root. The block of data sent from the j-th process is				
	received by every process and placed in the j-th block of the buffer recvbuf. These blocks need not all be the same size.			
need not a	lieed not all be the same size. 44			

The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[j], recvtype at any other process.

If comm is an intracommunicator, the outcome is as if all processes executed calls to

45

46

1 2	MPI_(Gatherv(sendbuf,sendcount	, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)
2 3 4 5 6 7 8 9 10 11 12 13 14	found from The " MPI_IN_PL sendtype a that proce If com sendcount in the oth processes	n the corresponding rules for "in place" option for intracom ACE to the argument sendbu- re ignored, and the input data ass would receive its own cont of is an intercommunicator, to data items; these data are co- ner group (group B). Convers- in group B is stored at each	nmunicators is specified by passing the value of at all processes. In such a case, sendcount and a of each process is assumed to be in the area where
15 16	6.7.1 Ex	ample using MPI_ALLGATH	ER
17	The exam	ple in this section uses intrace	ommunicators.
18 19 20		6.14 The all-gather version) int s from every process in t	of Example 6.2. Using MPI_ALLGATHER, we will he group to every process.
21 22 23 24 25 26 27	<pre>MPI_Comm comm; int gsize,sendarray[100]; int *rbuf; MPI_Comm_size(comm, &gsize); rbuf = (int *)malloc(gsize*100*sizeof(int));</pre>		
28 29 30	MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm); After the call, every process has the group-wide concatenation of the sets of data.		
31 32 33	6.8 All	-to-All Scatter/Gather	
34 35	MPL ALLT	FOALL (sendbuf, sendcount, ser	ndtype, recvbuf, recvcount, recvtype, comm)
36	- IN	sendbuf	starting address of send buffer (choice)
37 38 39	IN	sendcount	number of elements sent to each process (non-negative integer)
40	IN	sendtype	data type of send buffer elements (handle)
41	OUT	recvbuf	address of receive buffer (choice)
42 43 44	IN	recvcount	number of elements received from any process (non-negative integer)
45	IN	recvtype	data type of receive buffer elements (handle)
46	IN	comm	communicator (handle)
47 48	C bindin	g	

int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, 2 void *recvbuf, int recvcount, MPI_Datatype recvtype, 3 MPI_Comm comm) Fortran 2008 binding MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 6 comm. ierror) TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf INTEGER, INTENT(IN) :: sendcount, recvcount 9 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 10 TYPE(*), DIMENSION(...) :: recvbuf 11 TYPE(MPI_Comm), INTENT(IN) :: comm 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 14Fortran binding 15MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 16COMM, IERROR) 17<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 18 19 MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process 20sends distinct data to each of the receivers. The j-th block sent from process i is received 21by process j and is placed in the i-th block of recvbuf. 22 The type signature associated with sendcount, sendtype, at a process must be equal to 23 the type signature associated with recvcount, recvtype at any other process. This implies 24that the amount of data sent must be equal to the amount of data received, pairwise between 25every pair of processes. As usual, however, the type maps may be different. 26If comm is an intracommunicator, the outcome is as if each process executed a send to 27each process (itself included) with a call to, 2829 MPI_Send(sendbuf+i· sendcount· extent(sendtype),sendcount,sendtype,i, ...), 30 31and a receive from every other process with a call to, 32 33 MPI_Recv(recvbuf+i· recvcount· extent(recvtype),recvcount,recvtype,i,...). 34 All arguments on all processes are significant. The argument comm must have identical 35 values on all processes. 36 The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE to 37 the argument sendbuf at *all* processes. In such a case, sendcount and sendtype are ignored. 38 The data to be sent is taken from the recvbuf and replaced by the received data. Data sent 39 and received must have the same type map as specified by recvcount and recvtype. 40 41 Rationale. For large MPI_ALLTOALL instances, allocating both send and receive 42buffers may consume too much memory. The "in place" option effectively halves the 43 application memory consumption and is useful in situations where the data to be sent 44will not be used by the sending process after the MPI_ALLTOALL exchange (e.g., in 45parallel Fast Fourier Transforms). (End of rationale.) 46 47Advice to implementors. Users may opt to use the "in place" option in order to 48

Unofficial Draft for Comment Only

	200	Ch	HAPTER 6. COLLECTIVE COMMUNICATION	
1 2 3		conserve memory. Quality MPI implementations should thus strive to minimize system buffering. (<i>End of advice to implementors.</i>)		
4 5 6 7	sends a me	If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.		
8 9 10 11 12 13 14	Advice to users. When a complete exchange is executed on an intercommunication domain, then the number of data items sent from processes in group A to processes in group B need not equal the number of items sent in the reverse direction. In particular, one can have unidirectional communication by specifying sendcount = 0 in the reverse direction. (<i>End of advice to users.</i>)			
15 16 17 18	MPI_ALLT	OALLV(sendbuf, sendcounts, so recvtype, comm)	displs, sendtype, recvbuf, recvcounts, rdispls,	
19	IN	sendbuf	starting address of send buffer (choice)	
20 21 22	IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank	
23 24 25 26	IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j	
27	IN	sendtype	data type of send buffer elements (handle)	
28	OUT	recvbuf	address of receive buffer (choice)	
29 30 31	IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank	
32 33 34 35	IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	
36	IN	recvtype	data type of receive buffer elements (handle)	
37	IN	comm	communicator (handle)	
 38 39 40 41 42 43 44 	C binding int MPI_A	<pre>lltoallv(const void *send</pre>	dbuf, const int sendcounts[], MPI_Datatype sendtype, void *recvbuf, [], const int rdispls[], e, MPI_Comm comm)	
45	Fortran 2	008 binding		
46	MPI_Allto	MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,		
47 48	m	rdispls, recvtype, comm, ierror)		
40	TYPE(TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf		

```
INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
            rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(*), DIMENSION(..) :: recvbuf
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for the send is specified by sdispls and the location of the placement of the data on the receive side is specified by rdispls.

If comm is an intracommunicator, then the j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. These blocks need not all have the same size.

The type signature associated with sendcounts[j], sendtype at process i must be equal to the type signature associated with recvcounts[i], recvtype at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

The outcome is as if each process sent a message to every other process with,

 $MPI_Send(sendbuf+sdispls[i] \cdot extent(sendtype), sendcounts[i], sendtype, i, ...),$

and received a message from every other process with a call to

MPI_Recv(recvbuf+rdispls[i] · extent(recvtype),recvcounts[i],recvtype,i,...).

All arguments on all processes are significant. The argument **comm** must have identical values on all processes.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE to the argument sendbuf at *all* processes. In such a case, sendcounts, sdispls and sendtype are ignored. The data to be sent is taken from the recvbuf and replaced by the received data. Data sent and received must have the same type map as specified by the recvcounts array and the recvtype, and is taken from the locations of the receive buffer specified by rdispls.

Advice to users. Specifying the "in place" option (which must be given on all processes) implies that the same amount and type of data is sent and received between any two processes in the group of the communicator. Different pairs of processes can exchange different amounts of data. Users must ensure that recvcounts[j] and recvtype on process i match recvcounts[i] and recvtype on process j. This symmetric exchange can be useful in applications where the data to be sent will not be used by the sending process after the MPI_ALLTOALLV exchange. (*End of advice to users.*)

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process

Unofficial Draft for Comment Only

 $46 \\ 47$

1 2	i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.		
3 4 5 6 7	<i>Rationale.</i> The definitions of MPI_ALLTOALL and MPI_ALLTOALLV give as much flexibility as one would achieve by specifying n independent, point-to-point communications, with two exceptions: all messages use the same datatype, and messages are scattered from (or gathered to) sequential storage. (<i>End of rationale.</i>)		
8 9 10 11 12 13	Advice to implementors. Although the discussion of collective communication in terms of point-to-point operation implies that each message is transferred directly from sender to receiver, implementations may use a tree communication pattern. Messages can be forwarded by intermediate nodes where they are split (for scatter) or concatenated (for gather), if this is more efficient. (<i>End of advice to implementors.</i>)		
14 15			
16 17 18	MPI_ALL	TOALLW(sendbuf, se recvtypes, com	ndcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, nm)
19	IN	sendbuf	starting address of send buffer (choice)
20 21	IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each
22 23 24 25 26	IN	sdispls	rank integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers)
27 28 29 30	IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)
31	OUT	recvbuf	address of receive buffer (choice)
32 33 34	IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank
35 36 37 38 39	IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)
40 41 42	IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)
43 44	IN	comm	communicator (handle)
45 46 47	C bindini int MPI	0	oid *sendbuf, const int sendcounts[],
48		const int so	<pre>displs[], const MPI_Datatype sendtypes[],</pre>

1 void *recvbuf, const int recvcounts[], const int rdispls[], 2 const MPI_Datatype recvtypes[], MPI_Comm comm) 3 Fortran 2008 binding 4 MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, 5 rdispls, recvtypes, comm, ierror) 6 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf 7 INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*) 9 TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*) 10 TYPE(*), DIMENSION(...) :: recvbuf 11 TYPE(MPI_Comm), INTENT(IN) :: comm 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 14Fortran binding 15MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS, 16RDISPLS, RECVTYPES, COMM, IERROR) 17<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*), 18 19 RDISPLS(*), RECVTYPES(*), COMM, IERROR 20MPI_ALLTOALLW is the most general form of complete exchange. Like 21MPI_TYPE_CREATE_STRUCT, the most general type constructor, MPI_ALLTOALLW al-22 lows separate specification of count, displacement and datatype. In addition, to allow max-23 imum flexibility, the displacement of blocks within the send and receive buffers is specified 24in bytes. 25If comm is an intracommunicator, then the j-th block sent from process i is received by 26

If comm is an intracommunicator, then the j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. These blocks need not all have the same size.

The type signature associated with sendcounts[j], sendtypes[j] at process i must be equal to the type signature associated with recvcounts[i], recvtypes[i] at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

The outcome is as if each process sent a message to every other process with

 MPI_Send(sendbuf+sdispls[i],sendcounts[i],sendtypes[i],i,...),
 35

 and received a message from every other process with a call to
 36

 MPI_Recv(recvbuf+rdispls[i],recvcounts[i],recvtypes[i],i,...).
 38

 All arguments on all processes are significant. The argument comm must describe the
 40

 same communicator on all processes.
 41

Like for MPI_ALLTOALLV, the "in place" option for intracommunicators is specified by passing MPI_IN_PLACE to the argument sendbuf at *all* processes. In such a case, sendcounts, sdispls and sendtypes are ignored. The data to be sent is taken from the recvbuf and replaced by the received data. Data sent and received must have the same type map as specified by the received must have the same type map as specified by the received and receives arrays, and is taken from the locations of the receive buffer specified by rdispls.

27

28

29

30

 31

32

33 34

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

Rationale. The MPI_ALLTOALLW function generalizes several MPI functions by carefully selecting the input arguments. For example, by making all but one process have sendcounts[i] = 0, this achieves an MPI_SCATTERW function. (*End of rationale.*)

6.9 Global Reduction Operations

The functions in this section perform a global reduce operation (for example sum, maximum, and logical and) across all members of a group. The reduction operation can be either one of a predefined list of operations, or a user-defined operation. The global reduction functions come in several flavors: a reduce that returns the result of the reduction to one member of a group, an all-reduce that returns this result to all members of a group, and two scan (parallel prefix) operations. In addition, a reduce-scatter operation combines the functionality of a reduce and of a scatter operation.

6.9.1 Reduce

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

		•	
25	IN	sendbuf	address of send buffer (choice)
26	OUT	recvbuf	address of receive buffer (choice, significant only at
27 28			root)
29	IN	count	number of elements in send buffer (non-negative
30			integer)
31	IN	datatype	data type of elements of send buffer (handle)
32 33	IN	ор	reduce operation (handle)
34	IN	root	rank of root process (integer)
35	IN	comm	communicator (handle)
36			
37	C binding		
38	<pre>int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,</pre>		
39 40	MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)		
40	Fortran 2008 binding		
42	MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)		
43	TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf		
44	TYPE(*), DIMENSION() :: recvbuf		
45	INTEGER, INTENT(IN) :: count, root		
46	TYPE(MPI_Datatype), INTENT(IN) :: datatype		
47	TYPE(MPI_Op), INTENT(IN) :: op		
48	TYPE(MPI_Comm), INTENT(IN) :: comm		

 $\mathbf{2}$

 $\overline{7}$

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

If comm is an intracommunicator, MPI_REDUCE combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root. The input buffer is defined by the arguments sendbuf, count and datatype; the output buffer is defined by the arguments recvbuf, count and datatype; both have the same number of elements, with the same type. The routine is called by all group members using the same arguments for count, datatype, op, root and comm. Thus, all processes provide input buffers of the same length, with elements of the same type as the output buffer at the root. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence. For example, if the operation is MPI_MAX and the send buffer contains two elements that are floating point numbers (count = 2 and datatype = MPI_FLOAT), then recvbuf(1) = global max(sendbuf(1)) and recvbuf(2) = global max(sendbuf(2)).

Section 6.9.2, lists the set of predefined operations provided by MPI. That section also enumerates the datatypes to which each operation can be applied.

In addition, users may define their own operations that can be overloaded to operate on several datatypes, either basic or derived. This is further explained in Section 6.9.5.

The operation **op** is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative, but not commutative. The "canonical" evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity in order to change the order of evaluation. This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition.

Advice to implementors. It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of ranks. (*End of advice to implementors.*)

Advice to users. Some applications may not be able to ignore the non-associative nature of floating-point operations or may use user-defined operations (see Section 6.9.5) that require a special reduction order and cannot be treated as associative. Such applications should enforce the order of evaluation explicitly. For example, in the case of operations that require a strict left-to-right (or right-to-left) evaluation order, this could be done by gathering all operands at a single process (e.g., with MPI_GATHER), applying the reduction operation in the desired order (e.g., with MPI_REDUCE_LOCAL), and if needed, broadcast or scatter the result to the other processes (e.g., with MPI_BCAST). (End of advice to users.)

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Section 6.9.2 and Section 6.9.4. Furthermore, the datatype and op given for predefined operators must be the same on all processes.

Unofficial Draft for Comment Only

 24

 $45 \\ 46$

Note that it is possible for users to supply different user-defined operations to
 MPI_REDUCE in each process. MPI does not define which operations are used on which
 operands in this case. User-defined operators may operate on general, derived datatypes.
 In this case, each argument that the reduce operation is applied to is one element described
 by such a datatype, which may contain several basic values. This is further explained in
 Section 6.9.5.

Advice to users. Users should make no assumptions about how MPI_REDUCE is implemented. It is safest to ensure that the same function is passed to MPI_REDUCE by each process. (*End of advice to users.*)

Overlapping datatypes are permitted in "send" buffers. Overlapping datatypes in "receive" buffers are erroneous and may give unpredictable results.

The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at the root. In such a case, the input data is taken at the root from the receive buffer, where it will be replaced by the output data.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Only send buffer arguments are significant in group B and only receive buffer arguments are significant at the root.

22 23

 24

7

8

9

10 11

12

13

14

15

16

17

18

19

20

21

6.9.2 Predefined Reduction Operations

The following predefined operations are supplied for MPI_REDUCE and related functions
 MPI_ALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER,
 MPI_SCAN, MPI_EXSCAN, all nonblocking variants of those (see Section 6.12), and
 MPI_REDUCE_LOCAL. These operations are invoked by placing the following in op.

29

30		
31	Name	Meaning
32		
33	MPI_MAX	maximum
34	MPI_MIN	minimum
35	MPI_SUM	sum
36	MPI_PROD	product
37	MPI_LAND	logical and
38	MPI_BAND	bit-wise and
39	MPI_LOR	logical or
10	MPI_BOR	bit-wise or
41	MPI_LXOR	logical exclusive or (xor)
12	MPI_BXOR	bit-wise exclusive or (xor)
13	MPI_MAXLOC	max value and location
	MPI_MINLOC	min value and location
14		

The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately in Section 6.9.4. For the other predefined operations, we enumerate below the allowed combinations of op and datatype arguments. First, define groups of MPI basic datatypes in the following way.

		1
C integer:	MPI_INT, MPI_LONG, MPI_SHORT,	2
-	MPI_UNSIGNED_SHORT, MPI_UNSIGNED,	3
	MPI_UNSIGNED_LONG,	4
	MPI_LONG_LONG_INT,	5
	MPI_LONG_LONG (as synonym),	6
	MPI_UNSIGNED_LONG_LONG,	7
	MPI_SIGNED_CHAR,	8
	MPI_UNSIGNED_CHAR,	9
	MPI_INT8_T, MPI_INT16_T,	10
	MPI_INT32_T, MPI_INT64_T,	11
	MPI_UINT8_T, MPI_UINT16_T,	12
	MPI_UINT32_T, and MPI_UINT64_T	13
Fortran integer:	MPI_INTEGER	14
	and handles returned from	15
	MPI_TYPE_CREATE_F90_INTEGER	16
	and, if available, MPI_INTEGER1,	17
	MPI_INTEGER2, MPI_INTEGER4,	18
	MPI_INTEGER8, and MPI_INTEGER16	19
Floating point:	MPI_FLOAT, MPI_DOUBLE, MPI_REAL,	20
	MPI_DOUBLE_PRECISION,	21
	MPI_LONG_DOUBLE,	22
	and handles returned from	23
	MPI_TYPE_CREATE_F90_REAL	24
	and, if available, MPI_REAL2,	25
т • 1	MPI_REAL4, MPI_REAL8, and MPI_REAL16	26
Logical:	MPI_LOGICAL, MPI_C_BOOL,	27
Complex	and MPI_CXX_BOOL	28
Complex:	MPI_COMPLEX, MPI_C_COMPLEX,	29
	MPI_C_FLOAT_COMPLEX (as synonym), MPI_C_DOUBLE_COMPLEX,	30
	MPI_C_DOUBLE_COMPLEX, MPI_C_LONG_DOUBLE_COMPLEX,	31
	MPI_CXX_FLOAT_COMPLEX,	32
	MPI_CXX_DOUBLE_COMPLEX,	33
	MPI_CXX_LONG_DOUBLE_COMPLEX,	34
	and handles returned from	35
	MPI_TYPE_CREATE_F90_COMPLEX	36
	and, if available, MPI_DOUBLE_COMPLEX,	
	MPI_COMPLEX4, MPI_COMPLEX8,	37
	MPI_COMPLEX16, and MPI_COMPLEX32	38
Byte:	MPI_BYTE	39
Multi-language types:	MPI_AINT, MPI_OFFSET, and MPI_COUNT	40
		41
Now, the valid datatypes for each o	peration are specified below.	42
		43
	Allowed Trunca	44
Ор	Allowed Types	45
	Cinterney Fortune interney Flasting solid	46
MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point,	47
	Multi-language types	48

Unofficial Draft for Comment Only

```
1
       MPI_SUM, MPI_PROD
                                              C integer, Fortran integer, Floating point, Complex,
\mathbf{2}
                                              Multi-language types
3
       MPI_LAND, MPI_LOR, MPI_LXOR
                                              C integer, Logical
4
       MPI_BAND, MPI_BOR, MPI_BXOR
                                              C integer, Fortran integer, Byte, Multi-language types
5
          These operations together with all listed datatypes are valid in all supported program-
6
     ming languages, see also Reduce Operations on page 785 in Section 19.2.6.
7
          The following examples use intracommunicators.
8
9
     Example 6.15 A routine that computes the dot product of two vectors that are distributed
10
     across a group of processes and returns the answer at node zero.
11
12
     SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
13
     REAL a(m), b(m)
                               ! local slice of array
14
     REAL c
                               ! result (at node zero)
15
     REAL sum
16
     INTEGER m, comm, i, ierr
17
18
     ! local sum
19
     sum = 0.0
20
     DO i = 1, m
21
         sum = sum + a(i)*b(i)
22
     END DO
23
^{24}
     ! global sum
25
     CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
26
     RETURN
27
     END
28
29
     Example 6.16 A routine that computes the product of a vector and an array that are
30
     distributed across a group of processes and returns the answer at node zero.
^{31}
32
     SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
33
     REAL a(m), b(m,n)
                              ! local slice of array
34
     REAL c(n)
                                result
35
     REAL sum(n)
36
     INTEGER n, comm, i, j, ierr
37
38
     ! local sum
39
     DO j=1,n
40
         sum(j) = 0.0
41
         DO i=1,m
42
            sum(j) = sum(j) + a(i)*b(i,j)
43
         END DO
44
     END DO
45
46
     ! global sum
47
     CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)
48
```

CHAPTER 6. COLLECTIVE COMMUNICATION

! return result at node zero (and garbage at the other nodes) RETURN END

6.9.3 Signed Characters and Reductions

The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR can be used in reduction operations. MPI_CHAR, MPI_WCHAR, and MPI_CHARACTER (which represent printable characters) cannot be used in reduction operations. In a heterogeneous environment, MPI_CHAR, MPI_WCHAR, and MPI_CHARACTER will be translated so as to preserve the printable character, whereas MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR will be translated so as to preserve the integer value.

Advice to users. The types MPI_CHAR, MPI_WCHAR, and MPI_CHARACTER are intended for characters, and so will be translated to preserve the printable representation, rather than the integer value, if sent between machines with different character codes. The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR should be used in C if the integer value should be preserved. (*End of advice to users.*)

6.9.4 MINLOC and MAXLOC

The operator MPI_MINLOC is used to compute a global minimum and also an index attached to the minimum value. MPI_MAXLOC similarly computes a global maximum and index. One application of these is to compute a global minimum (maximum) and the rank of the process containing this value.

The operation that defines MPI_MAXLOC is:

$$\left(\begin{array}{c} u\\i\end{array}\right)\circ\left(\begin{array}{c} v\\j\end{array}\right)=\left(\begin{array}{c} w\\k\end{array}\right)$$

where

$$w = \max(u, v)$$

and

k

$$= \begin{cases} i & \text{if } u > v \\ \min(i,j) & \text{if } u = v \\ i & \text{if } u < v \end{cases}$$

MPI_MINLOC is defined similarly:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$
⁴¹
⁴²
⁴³

where

$$w = \min(u, v)$$

Unofficial Draft for Comment Only

 24

and

1

ſ	i	if $u < v$
$k = \left\{ \right.$	$\min(i, j)$	if $u = v$
l	j	if u > v

6 Both operations are associative and commutative. Note that if MPI_MAXLOC is applied 7to reduce a sequence of pairs $(u_0,0), (u_1,1), \ldots, (u_{n-1},n-1)$, then the value returned is 8 (u, r), where $u = \max_i u_i$ and r is the index of the first global maximum in the sequence. 9 Thus, if each process supplies a value and its rank within the group, then a reduce operation 10 with $op = MPI_MAXLOC$ will return the maximum value and the rank of the first process with 11that value. Similarly, MPI_MINLOC can be used to return a minimum and its index. More 12generally, MPI_MINLOC computes a *lexicographic minimum*, where elements are ordered 13according to the first component of each pair, and ties are resolved according to the second 14component.

¹⁵ The reduce operation is defined to operate on arguments that consist of a pair: value ¹⁶ and index. For both Fortran and C, types are provided to describe the pair. The potentially ¹⁷ mixed-type nature of such arguments is a problem in Fortran. The problem is circumvented, ¹⁸ for Fortran, by having the MPI-provided type consist of a pair of the same type as value, ¹⁹ and coercing the index to this type also. In C, the MPI-provided pair type has distinct ²⁰ types and the index is an int.

In order to use MPI_MINLOC and MPI_MAXLOC in a reduce operation, one must provide
 a datatype argument that represents a pair (value and index). MPI provides nine such
 predefined datatypes. The operations MPI_MAXLOC and MPI_MINLOC can be used with
 each of the following datatypes.

²⁶ Fortran:

	10101000		
27	Name	Description	
28	MPI_2REAL	pair of REALs	
29	MPI_2DOUBLE_PRECISION	pair of DOUBLE PRECISION variables	
30	MPI_2INTEGER	pair of INTEGERS	
31			
32			
33	C:		
34	Name	Description	
35	MPI_FLOAT_INT	float and int	
36	MPI_DOUBLE_INT	double and int	
37	MPI_LONG_INT	long and int	
38	MPI_2INT	pair of int	
39	MPI_SHORT_INT	short and int	
40	MPI_LONG_DOUBLE_INT	long double and int	
41	The datatype MPL 2RFAL is a	s if defined by the following (see Section 5.1)	
42	The datatype MPI_2REAL is as if defined by the following (see Section 5.1).		
43	MPI_Type_contiguous(2, MPI_REAL, MPI_2REAL);		
44		,,	
45	Similar statements apply for MPI_2INTEGER, MPI_2DOUBLE_PRECISION, and MPI_2INT		
46		Γ is as if defined by the following sequence of instructions.	
47			
48			

```
1
struct mystruct {
                                                                                          2
    short val;
                                                                                          3
    int rank;
};
                                                                                          4
type[0] = MPI_SHORT;
                                                                                          5
                                                                                          6
type[1] = MPI_INT;
disp[0] = 0;
disp[1] = offsetof(struct mystruct, rank);
block[0] = 1;
block[1] = 1;
                                                                                          10
                                                                                          11
MPI_Type_create_struct(2, block, disp, type, MPI_SHORT_INT);
                                                                                          12
Similar statements apply for MPI_FLOAT_INT, MPI_LONG_INT and MPI_DOUBLE_INT.
                                                                                          13
    The following examples use intracommunicators.
                                                                                          14
                                                                                          15
Example 6.17 Each process has an array of 30 doubles, in C. For each of the 30 locations,
                                                                                          16
compute the value and rank of the process containing the largest value.
                                                                                          17
                                                                                          18
    . . .
                                                                                          19
    /* each process has an array of 30 double: ain[30]
                                                                                          20
     */
                                                                                         21
    double ain[30], aout[30];
                                                                                          22
    int ind[30];
                                                                                          23
    struct {
                                                                                          ^{24}
        double val;
                                                                                          25
         int
               rank;
                                                                                          26
    } in[30], out[30];
                                                                                          27
    int i, myrank, root;
                                                                                          28
                                                                                          29
    MPI_Comm_rank(comm, &myrank);
                                                                                          30
    for (i=0; i<30; ++i) {</pre>
                                                                                          31
         in[i].val = ain[i];
                                                                                          32
         in[i].rank = myrank;
                                                                                          33
                                                                                         34
    MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
                                                                                         35
    /* At this point, the answer resides on process root
                                                                                          36
     */
                                                                                          37
    if (myrank == root) {
                                                                                          38
         /* read ranks out
                                                                                          39
          */
                                                                                          40
        for (i=0; i<30; ++i) {</pre>
                                                                                          41
             aout[i] = out[i].val;
                                                                                          42
             ind[i] = out[i].rank;
                                                                                          43
        }
                                                                                          44
    }
                                                                                          45
                                                                                          46
```

Example 6.18 Same example, in Fortran.

47

```
1
      . . .
\mathbf{2}
     ! each process has an array of 30 double: ain(30)
3
4
     DOUBLE PRECISION ain(30), aout(30)
\mathbf{5}
     INTEGER ind(30)
6
     DOUBLE PRECISION in(2,30), out(2,30)
7
     INTEGER i, myrank, root, ierr
8
9
     CALL MPI_COMM_RANK(comm, myrank, ierr)
10
     DO i=1,30
^{11}
         in(1,i) = ain(i)
12
         in(2,i) = myrank
                               ! myrank is coerced to a double
13
     END DO
14
15
     CALL MPI_REDUCE(in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root,&
16
                       comm, ierr)
17
     ! At this point, the answer resides on process root
^{18}
19
     IF (myrank .EQ. root) THEN
20
         ! read ranks out
21
        DO i=1,30
22
            aout(i) = out(1,i)
23
            ind(i) = out(2,i) ! rank is coerced back to an integer
^{24}
        END DO
25
     END IF
26
27
     Example 6.19 Each process has a non-empty array of values. Find the minimum global
28
     value, the rank of the process that holds it and its index on this process.
29
30
     #define LEN
                      1000
31
32
     float val[LEN];
                               /* local array of values */
33
                               /* local number of values */
     int count;
34
     int myrank, minrank, minindex;
35
     float minval;
36
37
     struct {
38
          float value;
39
          int
                 index;
40
     } in, out;
41
42
          /* local minloc */
43
     in.value = val[0];
44
     in.index = 0;
45
     for (i=1; i < count; i++)</pre>
46
          if (in.value > val[i]) {
47
              in.value = val[i];
48
```

```
1
         in.index = i;
                                                                                           \mathbf{2}
    }
                                                                                           3
    /* global minloc */
MPI_Comm_rank(comm, &myrank);
                                                                                           5
in.index = myrank*LEN + in.index;
                                                                                           6
MPI_Reduce(&in, &out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm);
    /* At this point, the answer resides on process root
      */
                                                                                           9
                                                                                           10
if (myrank == root) {
                                                                                           11
    /* read answer out
     */
                                                                                           12
                                                                                           13
    minval = out.value;
                                                                                           14
    minrank = out.index / LEN;
                                                                                           15
    minindex = out.index % LEN;
                                                                                           16
}
                                                                                           17
     Rationale.
                   The definition of MPI_MINLOC and MPI_MAXLOC given here has the
                                                                                          18
     advantage that it does not require any special-case handling of these two operations:
                                                                                          19
     they are handled like any other reduce operation. A programmer can provide his or
                                                                                          20
     her own definition of MPI_MAXLOC and MPI_MINLOC, if so desired. The disadvantage
                                                                                          21
     is that values and indices have to be first interleaved, and that indices and values have
                                                                                          22
     to be coerced to the same type, in Fortran. (End of rationale.)
                                                                                          23
                                                                                           ^{24}
                                                                                           25
6.9.5 User-Defined Reduction Operations
                                                                                           26
                                                                                          27
                                                                                           28
MPI_OP_CREATE(user_fn, commute, op)
                                                                                          29
  IN
            user_fn
                                        user defined function (function)
                                                                                           30
  IN
                                        true if commutative; false otherwise.
           commute
                                                                                           31
                                                                                           32
  OUT
                                        operation (handle)
            op
                                                                                           33
                                                                                          34
C binding
                                                                                          35
int MPI_Op_create(MPI_User_function *user_fn, int commute, MPI_Op *op)
                                                                                          36
                                                                                          37
Fortran 2008 binding
MPI_Op_create(user_fn, commute, op, ierror)
                                                                                           38
                                                                                           39
    PROCEDURE(MPI_User_function) :: user_fn
    LOGICAL, INTENT(IN) :: commute
                                                                                           40
                                                                                           41
    TYPE(MPI_Op), INTENT(OUT) :: op
                                                                                           42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                           43
Fortran binding
                                                                                           44
MPI_OP_CREATE(USER_FN, COMMUTE, OP, IERROR)
                                                                                           45
    EXTERNAL USER_FN
                                                                                           46
    LOGICAL COMMUTE
                                                                                           47
    INTEGER OP, IERROR
                                                                                           48
```

1	MPI_OP_CREATE binds a user-defined reduction operation to an
2	op handle that can subsequently be used in MPI_REDUCE, MPI_ALLREDUCE,
3	MPI_REDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_SCAN,
4	MPI_EXSCAN, all nonblocking variants of those (see Section 6.12), and
5	MPI_REDUCE_LOCAL. The user-defined operation is assumed to be associative. If commute
	*
6	= true, then the operation should be both commutative and associative. If commute = false,
7	then the order of operands is fixed and is defined to be in ascending, process rank order,
8	beginning with process zero. The order of evaluation can be changed, talking advantage of
9	the associativity of the operation. If $commute = true$ then the order of evaluation can be
10	changed, taking advantage of commutativity and associativity.
11	The argument user_fn is the user-defined function, which must have the following four
12	arguments: invec, inoutvec, len, and datatype.
13	The ISO C prototype for the function is the following.
14	typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
15	
16	<pre>MPI_Datatype *datatype);</pre>
	The Fortran declarations of the user-defined function user_fn appear below.
17	ABSTRACT INTERFACE
18	SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype)
19	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
20	TYPE(C_PTR), VALUE :: invec, inoutvec
21	INTEGER :: len
22	
23	TYPE(MPI_Datatype) :: datatype
24	SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
25	<type> INVEC(LEN), INOUTVEC(LEN)</type>
26	INTEGER LEN, DATATYPE
27	
28	The datatype argument is a handle to the data type that was passed into the call to
29	MPI_REDUCE. The user reduce function should be written such that the following holds:
30	Let u[0],, u[len-1] be the len elements in the communication buffer described by the
31	arguments invec, len and datatype when the function is invoked; let $v[0], \ldots, v[len-1]$ be len
32	elements in the communication buffer described by the arguments inoutvec, len and datatype
	when the function is invoked; let $w[0], \ldots, w[len-1]$ be len elements in the communication
33	buffer described by the arguments inoutvec, len and datatype when the function returns;
34	then $w[i] = u[i] \circ v[i]$, for i=0,, len-1, where \circ is the reduce operation that the function
35	
36	computes.
37	Informally, we can think of invec and inoutvec as arrays of len elements that user_fn
38	is combining. The result of the reduction over-writes values in inoutvec, hence the name.
39	Each invocation of the function results in the pointwise evaluation of the reduce operator
40	on len elements: i.e., the function returns in $inoutvec[i]$ the value $invec[i] \circ inoutvec[i]$, for
41	i=0,, count-1, where \circ is the combining operation computed by the function.
42	
43	Rationale. The len argument allows MPI_REDUCE to avoid calling the function for
44	each element in the input buffer. Rather, the system can choose to apply the function
45	to chunks of input. In C, it is passed in as a reference for reasons of compatibility
46	with Fortran.
	By internally comparing the value of the datatype argument to known, global handles,
47	
48	it is possible to overload the use of a single user-defined function for several, different

6.9. GLOBAL REDUCTION OPERATIONS 215	215
data types. (End of rationale.)	
General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies. No MPI communication function may be called inside the user function. MPI_ABORT may be called inside the function in case of an error.	
Advice to users. Suppose one defines a library of user-defined reduce functions that are overloaded: the datatype argument is used to select the right execution path at each invocation, according to the types of the operands. The user-defined reduce function cannot "decode" the datatype argument that it is passed, and cannot identify, by itself, the correspondence between the datatype handles and the datatype they represent. This correspondence was established when the datatypes were created. Before the library is used, a library initialization preamble must be executed. This preamble code will define the datatypes that are used by the library, and store handles to these datatypes in global, static variables that are shared by the user code and the library code.	ect the right execution path at each The user-defined reduce function ssed, and cannot identify, by itself, and the datatype they represent. tatypes were created. Before the nust be executed. This preamble library, and store handles to these d by the user code and the library
The Fortran version of MPI_REDUCE will invoke a user-defined reduce function using the Fortran calling conventions and will pass a Fortran-type datatype argument; the C version will use C calling convention and the C representation of a datatype handle. Users who plan to mix languages should define their reduction functions accordingly. (<i>End of advice to users.</i>)	tran-type datatype argument; the presentation of a datatype handle.
Advice to implementors. We outline below a naive and inefficient implementation of MPI_REDUCE not supporting the "in place" option.	_
<pre>MPI_Comm_size(comm, &groupsize); MPI_Comm_rank(comm, &rank); if (rank > 0) { MPI_Recv(tempbuf, count, datatype, rank-1,); User_reduce(tempbuf, sendbuf, count, datatype);</pre>	
<pre>} if (rank < groupsize-1) { MPI_Send(sendbuf, count, datatype, rank+1,); }</pre>	e, rank+1,);
<pre>/* answer now resides in process groupsize-1 now send to root */ if (rank == root) {</pre>	psize-1 now send to root
<pre>MPI_Irecv(recvbuf, count, datatype, groupsize-1,, &req); }</pre>	pe, groupsize-1,, &req);
<pre>if (rank == groupsize-1) {</pre>	

MPI_Send(sendbuf, count, datatype, root, ...);

}

}

if (rank == root) {

MPI_Wait(&req, &status);

 $\mathbf{2}$

 $\mathbf{5}$

 $\overline{7}$

 21

 23 24

 41

```
1
           The reduction computation proceeds, sequentially, from process 0 to process
2
           groupsize-1. This order is chosen so as to respect the order of a possibly non-
3
           commutative operator defined by the function User_reduce(). A more efficient im-
4
           plementation is achieved by taking advantage of associativity and using a logarithmic
5
           tree reduction. Commutativity can be used to advantage, for those cases in which
6
           the commute argument to MPI_OP_CREATE is true. Also, the amount of temporary
7
           buffer required can be reduced, and communication can be pipelined with computa-
8
           tion, by transferring and reducing the elements in chunks of size len <count.
9
           The predefined reduce operations can be implemented as a library of user-defined
10
           operations. However, better performance might be achieved if MPI_REDUCE handles
11
           these functions as a special case. (End of advice to implementors.)
12
13
14
     MPI_OP_FREE(op)
15
16
       INOUT
                                             operation (handle)
                 op
17
18
     C binding
19
     int MPI_Op_free(MPI_Op *op)
20
21
     Fortran 2008 binding
     MPI_Op_free(op, ierror)
22
          TYPE(MPI_Op), INTENT(INOUT) :: op
23
24
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     Fortran binding
26
     MPI_OP_FREE(OP, IERROR)
27
          INTEGER OP, IERROR
28
29
          Marks a user-defined reduction operation for deallocation and sets op to MPI_OP_NULL.
30
^{31}
     Example of User-Defined Reduce
32
     It is time for an example of user-defined reduction. The example in this section uses an
33
     intracommunicator.
34
35
     Example 6.20 Compute the product of an array of complex numbers, in C.
36
37
     typedef struct {
38
          double real, imag;
39
     } Complex;
40
41
     /* the user-defined function
42
      */
43
     void myProd(void *inP, void *inoutP, int *len, MPI_Datatype *dptr)
44
     ſ
45
          int i;
46
          Complex c;
47
          Complex *in = (Complex *)inP, *inout = (Complex *)inoutP;
48
```

CHAPTER 6. COLLECTIVE COMMUNICATION

```
1
    for (i=0; i< *len; ++i) {</pre>
                                                                                      2
        c.real = inout->real*in->real -
                                                                                      3
                    inout->imag*in->imag;
        c.imag = inout->real*in->imag +
                                                                                      4
                    inout->imag*in->real;
                                                                                      5
                                                                                      6
        *inout = c;
        in++; inout++;
                                                                                      7
    }
}
                                                                                      9
                                                                                      10
                                                                                      11
/* and, to call it...
 */
                                                                                      12
                                                                                      13
. . .
                                                                                      14
    /* each process has an array of 100 Complexes
                                                                                      15
                                                                                      16
     */
                                                                                      17
    Complex a[100], answer[100];
                                                                                      18
    MPI_Op myOp;
                                                                                      19
    MPI_Datatype ctype;
                                                                                      20
                                                                                      21
    /* explain to MPI how type Complex is defined
                                                                                      22
     */
    MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
                                                                                      23
                                                                                      ^{24}
    MPI_Type_commit(&ctype);
                                                                                      25
    /* create the complex-product user-op
                                                                                      26
     */
    MPI_Op_create(myProd, 1, &myOp);
                                                                                      27
                                                                                      28
                                                                                      29
    MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
                                                                                      30
    /* At this point, the answer, which consists of 100 Complexes,
                                                                                      31
                                                                                      32
     * resides on process root
                                                                                      33
                                                                                      34
                                                                                      35
Example 6.21 How to use the mpi_f08 interface of the Fortran MPI_User_function.
                                                                                      36
                                                                                      37
subroutine my_user_function(invec, inoutvec, len, type)
                                                               bind(c)
                                                                                      38
   use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
                                                                                      39
   use mpi_f08
                                                                                      40
   type(c_ptr), value :: invec, inoutvec
                                                                                      41
   integer :: len
                                                                                      42
   type(MPI_Datatype) :: type
   real, pointer :: invec_r(:), inoutvec_r(:)
                                                                                      43
                                                                                      44
   if (type%MPI_VAL == MPI_REAL%MPI_VAL) then
      call c_f_pointer(invec, invec_r, (/ len /))
                                                                                      45
                                                                                      46
      call c_f_pointer(inoutvec, inoutvec_r, (/ len /))
                                                                                      47
      inoutvec_r = invec_r + inoutvec_r
                                                                                      48
   end if
```

1	end subroutine					
2 3	6.9.6 All-Reduce					
4						
5		-	erations where the result is returned to all processes			
6 7		s receive identical results.	cesses from the same group participating in these			
8	oporation					
9 10	MPI_ALLF	REDUCE(sendbuf, recvbuf, cou	int, datatype, op, comm)			
11	IN	sendbuf	starting address of send buffer (choice)			
12	OUT	recvbuf	starting address of receive buffer (choice)			
13 14 15	IN	count	number of elements in send buffer (non-negative integer)			
16	IN	datatype	data type of elements of send buffer (handle)			
17	IN	ор	operation (handle)			
18	IN	comm	communicator (handle)			
19						
20 21	C bindin	ıg				
22	int MPI_		ndbuf, void *recvbuf, int count,			
23		MPI_Datatype dataty	pe, MPI_Op op, MPI_Comm comm)			
24	Fortran 2	2008 binding				
25 26			count, datatype, op, comm, ierror)			
20		<pre>(*), DIMENSION(), INTEN (*), DIMENSION() :: rec</pre>				
28		GER, INTENT(IN) :: count	svbul			
29		(MPI_Datatype), INTENT(IN	J) :: datatype			
30		(MPI_Op), INTENT(IN) :: c				
31 32		(MPI_Comm), INTENT(IN) ::				
33	INTE	GER, OPTIONAL, INTENT(OUT	C) :: ierror			
34	Fortran 1	binding				
35			COUNT, DATATYPE, OP, COMM, IERROR)			
36	• -	e> SENDBUF(*), RECVBUF(*)				
37		GER COUNT, DATATYPE, OP,				
38 39			MPI_ALLREDUCE behaves the same as			
40	MPI_RED	UCE except that the result ap	pears in the receive buffer of all the group members.			
41	Adv	Advice to implementors. The all-reduce operations can be implemented as a re-				
42		*	owever, a direct implementation can lead to better			
43	perf	formance. (End of advice to in	nplementors.)			
44		··· 1 ··· C ···				
45 46		The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. In this case, the input data is				
40	taken at each process from the receive buffer, where it will be replaced by the output data.					
48						

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in group A is stored at each process in group B, and vice versa. Both groups should provide count and datatype arguments that specify the same type signature.

The following example uses an intracommunicator.

Example 6.22 A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 6.16).

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
                      ! local slice of array
REAL a(m), b(m,n)
REAL c(n)
                      ! result
REAL sum(n)
INTEGER n, comm, i, j, ierr
! local sum
DO j=1,n
   sum(j) = 0.0
   DO i=1,m
      sum(j) = sum(j) + a(i)*b(i,j)
   END DO
END DO
! global sum
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)
! return result at all nodes
RETURN
END
6.9.7
      Process-Local Reduction
```

The functions in this section are of importance to library implementors who may want to implement special reduction patterns that are otherwise not easily covered by the standard MPI operations.

The following function applies a reduction operator to local arguments.

1

 $\mathbf{2}$

3

4

5

6

7

8 9

10

11

12

13

14 15

16

17

18

19

20

21

22 23

 24

25 26

27

28

29 30

31 32

33

34

35

1 MPI_REDUCE_LOCAL(inbuf, inoutbuf, count, datatype, op) 2 IN inbuf input buffer (choice) 3 INOUT inoutbuf combined input and output buffer (choice) 4 5IN number of elements in inbuf and inoutbuf buffers count 6 (non-negative integer) 7 IN datatype data type of elements of inbuf and inoutbuf buffers 8 (handle) 9 operation (handle) IN ор 10 11 C binding 12int MPI_Reduce_local(const void *inbuf, void *inoutbuf, int count, 13 14MPI_Datatype datatype, MPI_Op op) 15Fortran 2008 binding 16MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror) 17TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf 18 TYPE(*), DIMENSION(..) :: inoutbuf 19 INTEGER, INTENT(IN) :: count 20TYPE(MPI_Datatype), INTENT(IN) :: datatype 21TYPE(MPI_Op), INTENT(IN) :: op 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23 24 Fortran binding MPI_REDUCE_LOCAL (INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR) 2526<type> INBUF(*), INOUTBUF(*) INTEGER COUNT, DATATYPE, OP, IERROR 2728The function applies the operation given by op element-wise to the elements of inbuf 29 and inoutbuf with the result stored element-wise in inoutbuf, as explained for user-defined 30 operations in Section 6.9.5. Both inbuf and inoutbuf (input as well as result) have the 31 same number of elements given by count and the same datatype given by datatype. The 32 MPI_IN_PLACE option is not allowed. 33 Reduction operations can be queried for their commutativity. 34 35 36 MPI_OP_COMMUTATIVE(op, commute) 37 IN operation (handle) op 38 OUT commute true if op is commutative, false otherwise (logical) 39 40 41 C binding 42int MPI_Op_commutative(MPI_Op op, int *commute) 43 Fortran 2008 binding 44 MPI_Op_commutative(op, commute, ierror) 45 TYPE(MPI_Op), INTENT(IN) :: op 46 LOGICAL, INTENT(OUT) :: commute 47 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

Fortran binding MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR) INTEGER OP, IERROR LOGICAL COMMUTE

6.10 Reduce-Scatter

MPI includes variants of the reduce operations where the result is scattered to all processes in a group on return. One variant scatters equal-sized blocks to all processes, while another variant scatters blocks that may vary in size for each process.

6.10.1 MPI_REDUCE_SCATTER_BLOCK

16MPI_REDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm) 17 IN sendbuf starting address of send buffer (choice) 18 OUT recvbuf starting address of receive buffer (choice) 19 20IN recvcount element count per block (non-negative integer) 21data type of elements of send and receive buffers IN datatype 22 (handle) 23IN operation (handle) 24op 25IN comm communicator (handle) 2627C binding 28 int MPI_Reduce_scatter_block(const void *sendbuf, void *recvbuf, 29 int recvcount, MPI_Datatype datatype, MPI_Op op, 30 MPI_Comm comm) 31 Fortran 2008 binding 32 33 MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm, ierror) 34 TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf 35TYPE(*), DIMENSION(..) :: recvbuf 36 INTEGER, INTENT(IN) :: recvcount 37 TYPE(MPI_Datatype), INTENT(IN) :: datatype 38 TYPE(MPI_Op), INTENT(IN) :: op 39 TYPE(MPI_Comm), INTENT(IN) :: comm 40 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 42Fortran binding 43 MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, 44 IERROR) 45<type> SENDBUF(*), RECVBUF(*) 46INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR 4748

1

 $\mathbf{2}$

5 6

9

10

11 12

1 If comm is an intracommunicator, MPI_REDUCE_SCATTER_BLOCK first performs a $\mathbf{2}$ global, element-wise reduction on vectors of $count = n^{*}recvcount$ elements in the send buffers 3 defined by sendbuf, count and datatype, using the operation op, where n is the number of 4 processes in the group of comm. The routine is called by all group members using the 5same arguments for recvcount, datatype, op and comm. The resulting vector is treated 6 as n consecutive blocks of recvcount elements that are scattered to the processes of the $\overline{7}$ group. The i-th block is sent to process i and stored in the receive buffer defined by recvbuf, 8 recvcount, and datatype.

Advice to implementors. The MPI_REDUCE_SCATTER_BLOCK routine is functionally equivalent to: an MPI_REDUCE collective operation with count equal to recvcount*n, followed by an MPI_SCATTER with sendcount equal to recvcount. However, a direct implementation may run faster. (*End of advice to implementors.*)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument on *all* processes. In this case, the input data is taken from the receive buffer.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in one group (group A) is scattered among processes in the other group (group B) and vice versa. Within each group, all processes provide the same value for the recvcount argument, and provide input vectors of count = $n^{recvcount}$ elements stored in the send buffers, where n is the size of the group. The number of elements count must be the same for the two groups. The resulting vector from the other group is scattered in blocks of recvcount elements among the processes in the group.

Rationale. The last restriction is needed so that the length of the send buffer of one group can be determined by the local recvcount argument of the other group. Otherwise, a communication is needed to figure out how many elements are reduced. (End of rationale.)

29 30 31

32

33

34

25

26

27

28

9

10

11

12

13 14

6.10.2 MPI_REDUCE_SCATTER

MPI_REDUCE_SCATTER extends the functionality of MPI_REDUCE_SCATTER_BLOCK such that the scattered blocks can vary in size. Block sizes are determined by the recvcounts array, such that the i-th block contains recvcounts[i] elements.

MPI_RED	UCE_SCATTER(sendbuf	, recvbuf, recvcounts, datatype, op, comm)	1
IN	sendbuf	starting address of send buffer (choice)	2
OUT	recvbuf	starting address of receive buffer (choice)	3 4
IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements of the result	5 6
		distributed to each process.	7
IN	datatype	data type of elements of send and receive buffers	8 9
		(handle)	10
IN	ор	operation (handle)	11
IN	comm	communicator (handle)	12
			13
C hindin			14

PI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, dataty	be, op,	со
--	---------	----

C binding

int	MPI_Reduce_scatter(const void *sendbuf, void *recvbuf,	
	<pre>const int recvcounts[], MPI_Datatype datatype, MPI_Op op,</pre>	
	MPI_Comm comm)	

Fortran 2008 binding

MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
ierror)
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION() :: recvbuf
<pre>INTEGER, INTENT(IN) :: recvcounts(*)</pre>
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

```
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
```

If comm is an intracommunicator, MPI_REDUCE_SCATTER first performs a global, element-wise reduction on vectors of $count = \sum_{i=0}^{n-1} recvcounts[i]$ elements in the send buffers defined by sendbuf, count and datatype, using the operation op, where n is the number of processes in the group of comm. The routine is called by all group members using the same arguments for recvcounts, datatype, op and comm. The resulting vector is treated as n consecutive blocks where the number of elements of the i-th block is recvcounts[i]. The blocks are scattered to the processes of the group. The i-th block is sent to process i and stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.

Advice to implementors. The MPI_REDUCE_SCATTER routine is functionally equivalent to: an MPI_REDUCE collective operation with count equal to the sum of recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. However, a direct implementation may run faster. (End of advice to implementors.)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer. It is not required to specify the "in place" option on all processes, since the processes for which recvcounts[i] == 0 may not have allocated a receive buffer.

⁵ If comm is an intercommunicator, then the result of the reduction of the data provided ⁶ by processes in one group (group A) is scattered among processes in the other group (group ⁷ B), and vice versa. Within each group, all processes provide the same recvcounts argument, ⁸ and provide input vectors of count = $\sum_{i=0}^{n-1} \text{recvcounts}[i]$ elements stored in the send buffers, ⁹ where n is the size of the group. The resulting vector from the other group is scattered in ¹⁰ blocks of recvcounts[i] elements among the processes in the group. The number of elements ¹¹ count must be the same for the two groups.

Rationale. The last restriction is needed so that the length of the send buffer can be determined by the sum of the local **recvcounts** entries. Otherwise, a communication is needed to figure out how many elements are reduced. (*End of rationale.*)

```
6.11 Scan
```

6.11.1 Inclusive Scan

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

24	IN	sendbuf	starting address of send buffer (choice)	
25	OUT	recvbuf	starting address of receive buffer (choice)	
26 27	IN	count	number of elements in input buffer (non-negative integer)	
28 29	IN	datatype	data type of elements of input buffer (handle)	
30	IN			
31		ор	operation (handle)	
32	IN	comm	communicator (handle)	
33				
34	C bindin	g		
35	int MPI_		void *recvbuf, int count,	
36		MPI_Datatype datatyp	e, MPI_Op op, MPI_Comm comm)	
37	Fortran 2	2008 binding		
38		9	datatype, op, comm, ierror)	
39	TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf			
40	TYPE	(*), DIMENSION() :: rec	vbuf	
41	INTE	GER, INTENT(IN) :: count		
42	TYPE(MPI_Datatype), INTENT(IN) :: datatype			
43	TYPE	(MPI_Op), INTENT(IN) :: og	p	
44	TYPE	(MPI_Comm), INTENT(IN) ::	comm	
45	INTE	GER, OPTIONAL, INTENT(OUT)) :: ierror	
46 47	Fortran	hinding		
47 48		5	DATATYPE, OP, COMM, IERROR)	

1

 $\mathbf{2}$

3

4

12

13

14

15 16 17

18 19

20 21 22

	<pre>/pe> SENDBUF(*), RE</pre>		1
INT	FEGER COUNT, DATATY	PE, OP, COMM, IERROR	2 3
If c	omm is an intracomm	unicator, MPI_SCAN is used to perform a prefix reduction on	4
	=	oup. The operation returns, in the receive buffer of the process	5
	,	the values in the send buffers of processes with ranks $0,\ldots,i$	6
		ed by all group members using the same arguments for count,	7
0 ¥		ept that for user-defined operations, the same rules apply as	8
		of operations supported, their semantics, and the constraints re as for MPI_REDUCE.	9
		intracommunicators is specified by passing MPI_IN_PLACE in	10
		is case, the input data is taken from the receive buffer, and	11
	by the output data.		12 13
-	• *	for intercommunicators.	13
			15
6.11.2	Exclusive Scan		16
			17
			18
MPI_EX	SCAN(sendbuf, recvbu	f, count, datatype, op, comm)	19
IN	sendbuf	starting address of send buffer (choice)	20 21
OUT	recvbuf	starting address of receive buffer (choice)	22
IN	count	number of elements in input buffer (non-negative	23
		integer)	24
IN	datatype	data type of elements of input buffer (handle)	25 26
IN	ор	operation (handle)	27
IN	comm	intracommunicator (handle)	28
			29
C bind	ing		30
int MP		l *sendbuf, void *recvbuf, int count,	31
	MPI_Datatype	e datatype, MPI_Op op, MPI_Comm comm)	32 33
Fortra	n 2008 binding		34
MPI_Ex:	scan(sendbuf, recvb	ouf, count, datatype, op, comm, ierror)	35
TYI	PE(*), DIMENSION(), INTENT(IN) :: sendbuf	36
	<pre>PE(*), DIMENSION(</pre>		37
	<pre>FEGER, INTENT(IN) :</pre>		38
	••	INTENT(IN) :: datatype	39
	PE(MPI_Op), INTENT(PE(MPI_Comm), INTEN	-	40
		ITENT(OUT) :: ierror	41
			42
	n binding		$43 \\ 44$
		BUF, COUNT, DATATYPE, OP, COMM, IERROR)	45
•	/pe> SENDBUF(*), RE	(PE, OP, COMM, IERROR	46
TN.	LUBR COUNT, DATAI	ing, or, contr, induction	47
			10

 48

1 If comm is an intracommunicator, MPI_EXSCAN is used to perform a prefix reduction $\mathbf{2}$ on data distributed across the group. The value in recvbuf on the process with rank 0 is 3 undefined, and recvbuf is not significant on process 0. The value in recvbuf on the process 4 with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes $\mathbf{5}$ with rank i > 1, the operation returns, in the receive buffer of the process with rank i, the 6 reduction of the values in the send buffers of processes with ranks $0, \ldots, i-1$ (inclusive). The $\overline{7}$ routine is called by all group members using the same arguments for count, datatype, op and 8 comm, except that for user-defined operations, the same rules apply as for MPI_REDUCE. 9 The type of operations supported, their semantics, and the constraints on send and receive 10 buffers, are as for MPI_REDUCE.

¹¹ The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in ¹² the sendbuf argument. In this case, the input data is taken from the receive buffer, and ¹³ replaced by the output data. The receive buffer on rank 0 is not changed by this operation. ¹⁴ This operation is invalid for intercommunicators.

Rationale. The exclusive scan is more general than the inclusive scan. Any inclusive scan operation can be achieved by using the exclusive scan and then locally combining the local contribution. Note that for non-invertable operations such as MPI_MAX, the exclusive scan cannot be computed with the inclusive scan. (*End of rationale.*)

6.11.3 Example using MPI_SCAN

 $_{23}$ The example in this section uses an intracommunicator.

Example 6.23 This example uses a user-defined operation to produce a segmented scan.A segmented scan takes, as input, a set of values and a set of logicals, and the logicals delineate the various segments of the scan. For example:

$values$ v_1	v_2	v_3 v	4	v_5	v_6	v_7	v_8
logicals 0	0	1 1	L	1	0	0	1
$result$ v_1	$v_1 + v_2$	v_3 v_3 -	$\vdash v_4 v_3$ -	$+v_4 + v_5$	v_6	$v_6 + v_7$	v_8

The operator that produces this effect is

$$\left(\begin{array}{c} u\\i\end{array}\right)\circ\left(\begin{array}{c} v\\j\end{array}\right)=\left(\begin{array}{c} w\\j\end{array}\right),$$

where

 $w = \begin{cases} u + v & \text{if } i = j \\ v & \text{if } i \neq j \end{cases}.$

Note that this is a non-commutative operator. C code that implements it is given below.

43 44 45

- 40
- 46 47
- 48

15

16

17

18 19

20 21

22

 24

25

26

37

38 39 40

41

```
typedef struct {
    double val;
    int log;
} SegScanPair;
/* the user-defined function
 */
void segScan(SegScanPair *in, SegScanPair *inout, int *len,
             MPI_Datatype *dptr)
{
    int i;
    SegScanPair c;
    for (i=0; i< *len; ++i) {</pre>
        if (in->log == inout->log)
            c.val = in->val + inout->val;
        else
            c.val = inout->val;
        c.log = inout->log;
        *inout = c;
        in++; inout++;
    }
}
```

Note that the inout argument to the user-defined function corresponds to the righthand operand of the operator. When using this operator, we must be careful to specify that it is non-commutative, as in the following.

```
28
int i, base;
                                                                                  29
SegScanPair a, answer;
                                                                                  30
MPI_Op
              myOp;
MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
                                                                                  31
                                                                                  32
MPI_Aint
              disp[2];
                                                                                  33
int
              blocklen[2] = { 1, 1};
                                                                                  34
MPI_Datatype sspair;
                                                                                  35
                                                                                  36
/* explain to MPI how type SegScanPair is defined
                                                                                  37
 */
MPI_Get_address(&a, disp);
                                                                                  38
                                                                                  39
MPI_Get_address(&a.log, disp+1);
                                                                                  40
base = disp[0];
                                                                                  41
for (i=0; i<2; ++i) disp[i] -= base;</pre>
                                                                                  42
MPI_Type_create_struct(2, blocklen, disp, type, &sspair);
MPI_Type_commit(&sspair);
                                                                                  43
                                                                                  44
/* create the segmented-scan user-op
                                                                                  45
 */
                                                                                  46
MPI_Op_create(segScan, 0, &myOp);
                                                                                  47
. . .
                                                                                  48
MPI_Scan(&a, &answer, 1, sspair, myOp, comm);
```

Unofficial Draft for Comment Only

1

2

3

45

6

7

9

10 11

12 13 14

15

16

17

18

19

20 21

22

23 24

25

26

1 2

7

6.12 Nonblocking Collective Operations

As described in Section 3.7, performance of many applications can be improved by over-3 lapping communication and computation, and many systems enable this. Nonblocking 4 collective operations combine the potential benefits of nonblocking point-to-point opera-5tions, to exploit overlap and to avoid synchronization, with the optimized implementation 6 and message scheduling provided by collective operations [34, 38]. One way of doing this would be to perform a blocking collective operation in a separate thread. An alternative 8 mechanism that often leads to better performance (e.g., avoids context switching, scheduler 9 overheads, and thread management) is to use nonblocking collective communication [36]. 10

The nonblocking collective communication model is similar to the model used for non-11 blocking point-to-point communication. A nonblocking call initiates a collective operation, 12which must be completed in a separate completion call. Once initiated, the operation 13 may progress independently of any computation or other communication at participating 14processes. In this manner, nonblocking collective operations can mitigate possible synchro-15nizing effects of collective operations by running them in the "background." In addition to 16enabling communication-computation overlap, nonblocking collective operations can per-17form collective operations on overlapping communicators, which would lead to deadlocks 18 with blocking operations. Their semantic advantages can also be useful in combination with 19point-to-point communication. 20

As in the nonblocking point-to-point case, all calls are local and return immediately, 21irrespective of the status of other processes. The call initiates the operation, which indicates 22that the system may start to copy data out of the send buffer and into the receive buffer. 23Once initiated, all associated send buffers and buffers associated with input arguments (such 24 as arrays of counts, displacements, or datatypes in the vector versions of the collectives) 25should not be modified, and all associated receive buffers should not be accessed, until the 26collective operation completes. The call returns a request handle, which must be passed to 27a completion call. 28

All completion calls (e.g., MPI_WAIT) described in Section 3.7.3 are supported for 29 nonblocking collective operations. Similarly to the blocking case, nonblocking collective 30 operations are considered to be complete when the local part of the operation is finished, 31 i.e., for the caller, the semantics of the operation are guaranteed and all buffers can be 32 safely accessed and modified. Completion does not indicate that other processes have 33 completed or even started the operation (unless otherwise implied by the description of 34the operation). Completion of a particular nonblocking collective operation also does not 35 indicate completion of any other posted nonblocking collective (or send-receive) operations, 36 whether they are posted before or after the completed operation. 37

- 38
- 39 40 41

42

Advice to users. Users should be aware that implementations are allowed, but not required (with exception of MPI_IBARRIER), to synchronize processes during the completion of a nonblocking collective operation. (End of advice to users.)

Upon returning from a completion call in which a nonblocking collective operation 43 completes, the values of the MPI_SOURCE and MPI_TAG fields in the associated status object, 44 if any, are undefined. The value of MPI_ERROR may be defined, if appropriate, according 45to the specification in Section 3.2.5. It is valid to mix different request types (i.e., any 46 combination of collective requests, I/O requests, generalized requests, or point-to-point 47requests) in functions that enable multiple completions (e.g., MPI_WAITALL). It is erroneous 48

to call MPI_REQUEST_FREE or MPI_CANCEL for a request associated with a nonblocking collective operation. Nonblocking collective requests created using the APIs described in this section are not persistent. However, persistent collective requests can be created using persistent collective operations described in Sections 6.13 and 8.8.

Rationale. Freeing an active nonblocking collective request could cause similar problems as discussed for point-to-point requests (see Section 3.7.3). Cancelling a request is not supported because the semantics of this operation are not well-defined. (End of rationale.)

Multiple nonblocking collective operations can be outstanding on a single communicator. If the nonblocking call causes some system resource to be exhausted, then it will fail and raise an error. Quality implementations of MPI should ensure that this happens only in pathological cases. That is, an MPI implementation should be able to support a large number of pending nonblocking operations.

Unlike point-to-point operations, nonblocking collective operations do not match with blocking collective operations, and collective operations do not have a tag argument. All processes must call collective operations (blocking and nonblocking) in the same order per communicator. In particular, once a process calls a collective operation, all other processes in the communicator must eventually call the same collective operation, and no other collective operation with the same communicator in between. This is consistent with 20the ordering rules for blocking collective operations in threaded environments.

Matching blocking and nonblocking collective operations is not allowed Rationale. because the implementation might use different communication algorithms for the two cases. Blocking collective operations may be optimized for minimal time to completion, while nonblocking collective operations may balance time to completion with CPU overhead and asynchronous progression.

The use of tags for collective operations can prevent certain hardware optimizations. (End of rationale.)

Advice to users. If program semantics require matching blocking and nonblocking collective operations, then a nonblocking collective operation can be initiated and immediately completed with a blocking wait to emulate blocking behavior. (End of advice to users.)

In terms of data movement, each nonblocking collective operation has the same effect as its blocking counterpart for intracommunicators and intercommunicators after completion. Likewise, upon completion, nonblocking collective reduction operations have the same effect as their blocking counterparts, and the same restrictions and recommendations on reduction orders apply.

The use of the "in place" option is allowed exactly as described for the corresponding blocking collective operations. When using the "in place" option, message buffers function as both send and receive buffers. Such buffers should not be modified or accessed until the operation completes.

Progression rules for nonblocking collective operations are similar to progression of nonblocking point-to-point operations, refer to Section 3.7.4.

46Advice to implementors. Nonblocking collective operations can be implemented with 47local execution schedules [37] using nonblocking point-to-point communication and a 48 reserved tag-space. (End of advice to implementors.)

Unofficial Draft for Comment Only

1 $\mathbf{2}$

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

2122

23

24

25

26

27

28

29 30

 31

32

33

34

35

36

37

38

39

40

41

42

43

44

1	6.12.1 N	onblocking Barrier Synchro	onization		
3					
4	MPI_IBAR	RIER(comm, request)			
5 6	IN	comm	communicator (handle)		
7	OUT	request	communication request (handle)		
8 9 10	C binding int MPI_I	g Ebarrier(MPI_Comm comm,	MPI_Request *request)		
11 12 13 14 15 16	MPI_Ibarr TYPE(TYPE(2008 binding rier(comm, request, ierr [MPI_Comm), INTENT(IN) : [MPI_Request), INTENT(OU ER, OPTIONAL, INTENT(OU	:: comm JT) :: request		
17 18 19 20		Dinding RIER(COMM, REQUEST, IERF ER COMM, REQUEST, IERRO			
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35	a process of dent of wh are enforce called MPI in the rem <i>Advi</i> dent can of mant mess	notifies that it has reached nether other processes have ed at the corresponding com- ator case will complete only _IBARRIER. In the intercor- ote group have called MPI_ <i>ce to users</i> . A nonblocking computations between the overlap the barrier latency a	barrier can be used to hide latency. Moving indepen- MPI_IBARRIER and the subsequent completion call and therefore shorten possible waiting times. The se- when mixing collective operations and point-to-point		
36					
37 38	MPI IRCA	ST(buffer, count, datatype, r	root, comm, request)		
39	INOUT	buffer	starting address of buffer (choice)		
40	IN	count	number of entries in buffer (non-negative integer)		
41 42	IN	datatype	data type of buffer (handle)		
43	IN	root	rank of broadcast root (integer)		
44 45	IN	comm	communicator (handle)		
46	OUT	request	communication request (handle)		
47 48	C binding	g			

```
1
int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype, int root,
                                                                                        2
              MPI_Comm comm, MPI_Request *request)
                                                                                        3
Fortran 2008 binding
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
                                                                                        5
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
                                                                                        6
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                        9
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                        10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        11
Fortran binding
                                                                                        12
                                                                                        13
MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)
                                                                                        14
    <type> BUFFER(*)
                                                                                        15
    INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR
                                                                                        16
    This call starts a nonblocking variant of MPI_BCAST (see Section 6.4).
                                                                                        17
                                                                                        18
Example using MPI_IBCAST
                                                                                        19
                                                                                        20
The example in this section uses an intracommunicator.
                                                                                        21
Example 6.24 Start a broadcast of 100 ints from process 0 to every process in the
                                                                                        22
                                                                                        23
group, perform some computation on independent data, and then complete the outstanding
                                                                                        ^{24}
broadcast operation.
                                                                                        25
                                                                                        26
    MPI_Comm comm;
    int array1[100], array2[100];
                                                                                        27
    int root=0;
                                                                                        28
    MPI_Request req;
                                                                                        29
                                                                                        30
    . . .
    MPI_Ibcast(array1, 100, MPI_INT, root, comm, &req);
                                                                                        31
    compute(array2, 100);
                                                                                        32
    MPI_Wait(&req, MPI_STATUS_IGNORE);
                                                                                        33
                                                                                        34
                                                                                        35
                                                                                        36
                                                                                        37
                                                                                        38
                                                                                        39
                                                                                        40
                                                                                        41
                                                                                        42
                                                                                        43
                                                                                        44
                                                                                        45
                                                                                        46
                                                                                        47
                                                                                        48
```

¹ 6.12.3	Nonblocking Gather	
3 4 MPI_IG 5	GATHER(sendbuf, sendco request)	unt, sendtype, recvbuf, recvcount, recvtype, root, comm,
6 7 IN	sendbuf	starting address of send buffer (choice)
8 IN 9	sendcount	number of elements in send buffer (non-negative integer)
¹⁰ IN	sendtype	data type of send buffer elements (handle)
11 12 OUT	recvbuf	address of receive buffer (choice, significant only at root)
14 IN 15	recvcount	number of elements for any single receive (non-negative integer, significant only at root)
¹⁶ IN	recvtype	data type of recv buffer elements (handle, significant only at root)
18 19 IN	root	rank of receiving process (integer)
20 IN	comm	communicator (handle)
21 OUT	request	communication request (handle)
29 MPI_Ig 30 1 TY 31 TY 32 IN 33 TY 34 TY 34 TY 35 TY 36 TY 37 IN 38 38 10 10	MPI_Comm comm an 2008 binding gather(sendbuf, sendo root, comm, n PE(*), DIMENSION() TEGER, INTENT(IN) :: PE(MPI_Datatype), IN	ENT(OUT) :: request
$^{40}_{40}$ MPI_IG	ATHER(SENDBUF, SENDC ROOT, COMM, F ype> SENDBUF(*), REC	COUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, REQUEST, IERROR) CVBUF(*) IDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
45 Th 46 47 48	iis call starts a nonblock	ting variant of MPI_GATHER (see Section 6.5).

MPI_IGAT	HERV(sendbuf, sendcount, s comm, request)	sendtype, recvbuf, recvcounts, displs, recvtype, root,	1 2	
IN	sendbuf	starting address of send buffer (choice)	3	
IN	sendcount	number of elements in send buffer (non-negative integer)	4 5 6	
IN	sendtype	data type of send buffer elements (handle)	7	
OUT	recvbuf	address of receive buffer (choice, significant only at root)	8 9 10	
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)	10 11 12 13	
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)	14 15 16 17	
IN	recvtype	data type of recv buffer elements (handle, significant only at root)	18 19 20	
IN	root	rank of receiving process (integer)	21	
IN	comm	communicator (handle)	22	
OUT	request	communication request (handle)	23 24	
C binding int MPI_I	gatherv(const void *se void *recvbuf, cor	ndbuf, int sendcount, MPI_Datatype sendtype, ast int recvcounts[], const int displs[], cype, int root, MPI_Comm comm, est)	25 26 27 28 29 30	
MPI_Igath TYPE(INTEG TYPE(INTEG TYPE(TYPE(TYPE(recvtype, root, co *), DIMENSION(), INT ER, INTENT(IN) :: send MPI_Datatype), INTENT(*), DIMENSION(), ASY	RONOUS :: recvcounts(*) :: comm UT) :: request	31 32 33 34 35 36 37 38 39 40 41 42	
Fortran binding MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, ROOT, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, COMM, REQUEST, IERROR</type>				

	234	CI	HAPTER 6. COLLECTIVE COMMUNICATION		
1 2	This call starts a nonblocking variant of $MPI_GATHERV$ (see Section 6.5).				
3 4 5	6.12.4 No	onblocking Scatter			
6 7 8	MPI_ISCAT	TTER(sendbuf, sendcount, send request)	dtype, recvbuf, recvcount, recvtype, root, comm,		
9 10	IN	sendbuf	address of send buffer (choice, significant only at root)		
11 12	IN	sendcount	number of elements sent to each process (non-negative integer, significant only at root)		
13 14 15	IN	sendtype	data type of send buffer elements (handle, significant only at root)		
16	OUT	recvbuf	address of receive buffer (choice)		
17 18	IN	recvcount	number of elements in receive buffer (non-negative integer)		
19 20	IN	recvtype	data type of receive buffer elements (handle)		
21	IN	root	rank of sending process (integer)		
22	IN	comm	communicator (handle)		
23 24	OUT	request	communication request (handle)		
25	C binding	r			
26 27	-		buf, int sendcount, MPI_Datatype sendtype,		
28 29	void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,				
30	Fortran 2	008 binding			
31	MPI_Iscat		sendtype, recvbuf, recvcount, recvtype,		
32 33		root, comm, request,			
34	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN) :: sendcount, recvcount, root				
35	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype				
36	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf				
37 38	TYPE(MPI_Comm), INTENT(IN) :: comm				
39	TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
40					
41 42	Fortran b		SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,		
42	ROOT, COMM, REQUEST, IERROR)				
44	<type> SENDBUF(*), RECVBUF(*)</type>				
45	INTEG	ER SENDCOUNT, SENDTYPE, H IERROR	RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,		
46 47					
48	This c	all starts a nonblocking varia	nt of $MPI_SCATTER$ (see Section 6.6).		

MPI_ISC/	ATTERV(sendbuf, sendco comm, request)	ounts, displs, sendtype, recvbuf, recvcount, recvtype, root,	1 2
IN	sendbuf	address of send buffer (choice, significant only at root)	3 4 5
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank (significant only at root)	6 7 8
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i (significant only at root)	9 10 11 12
IN	sendtype	data type of send buffer elements (handle, significant only at root)	13 14 15
OUT	recvbuf	address of receive buffer (choice)	16
IN	recvcount	number of elements in receive buffer (non-negative integer)	17 18
IN	recvtype	data type of receive buffer elements (handle)	19 20
IN	root	rank of sending process (integer)	20
IN	comm	communicator (handle)	22
OUT	request	communication request (handle)	23
			24
C bindi	ng		25 26
int MPI_	Iscatterv(const void	<pre>! *sendbuf, const int sendcounts[],</pre>	27
		<pre>ls[], MPI_Datatype sendtype, void *recvbuf,</pre>	28
		MPI_Datatype recvtype, int root, MPI_Comm comm,	29
	MPI_Request *re	equest)	30
	2008 binding		31
MPI_Isca		counts, displs, sendtype, recvbuf, recvcount,	32 33
TUDI		, comm, request, ierror)	33 34
		INTENT(IN), ASYNCHRONOUS :: sendbuf YNCHRONOUS :: sendcounts(*)	35
		displs(*), recvcount, root	36
		ENT(IN) :: sendtype, recvtype	37
		ASYNCHRONOUS :: recvbuf	38
TYPE	E(MPI_Comm), INTENT(I	IN) :: comm	39
	E(MPI_Request), INTEN	-	40
INTE	EGER, OPTIONAL, INTEN	NT(OUT) :: ierror	41 42
Fortran	binding		43
MPI_ISCA	TTERV(SENDBUF, SENDC	COUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,	44
		, COMM, REQUEST, IERROR)	45
• 1	<pre>> SENDBUF(*), RECVE</pre>		46
TN.LF	GER SENDCOUNTS(*), DCOMM, REQUEST,	DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,	47
	COURT, REQUESI,	, TRUMUK	48

236		CHAPTER 6.	COLLECTIVE COMMUNICATION
1 Thi	s call starts a nonblocki	ng variant of MPI_SC	CATTERV (see Section 6.6).
4	Nonblocking Gather-to	-all	
⁵ 7 MPI_IAL	LGATHER(sendbuf, send request)	dcount, sendtype, recv	vbuf, recvcount, recvtype, comm,
8 9 IN	sendbuf	starting add	lress of send buffer (choice)
10 IN 11	sendcount	-	lements in send buffer (non-negative
12 13	sendtype	data type of	f send buffer elements (handle)
13 14 OUT	recvbuf	address of re	eceive buffer (choice)
15 IN 16	recvcount	number of el (non-negativ	lements received from any process we integer)
17 18	recvtype	data type of	f receive buffer elements (handle)
19 IN	comm	communicat	or (handle)
20 OUT	request	communicat	ion request (handle)
<pre>int MPI_Iallgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) Fortran 2008 binding MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN) :: sendcount, recvcount TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, REQUEST, IERROR This call starts a nonblocking variant of MPI_ALLGATHER (see Section 6.7).</type></pre>			

MPI_IALI	LGATHERV(sendbuf, ser comm, request)	ndcount, sendtype, recvbuf, recvcounts, displs, recvtype,	1 2		
IN	sendbuf	starting address of send buffer (choice)	3		
IN	sendcount	number of elements in send buffer (non-negative integer)	4 5 6		
IN	sendtype	data type of send buffer elements (handle)	7		
OUT	recvbuf	address of receive buffer (choice)	8 9		
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process	10 11 12		
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	13 14 15		
IN	recvtype	data type of receive buffer elements (handle)	16 17		
IN	comm	communicator (handle)	18		
OUT	request	communication request (handle)	19		
			20 21		
C bindi	0	oid *sendbuf int sendcount	22		
<pre>int MPI_Iallgatherv(const void *sendbuf, int sendcount,</pre>					
		ols[], MPI_Datatype recvtype, MPI_Comm comm,	24 25		
MPI_Request *request)					
	Fortran 2008 binding				
MPI_Iall		ndcount, sendtype, recvbuf, recvcounts, displs,	28		
TVDE		, request, ierror) INTENT(IN), ASYNCHRONOUS :: sendbuf	29		
	EGER, INTENT(IN) ::		30 31		
		ENT(IN) :: sendtype, recvtype	32		
		ASYNCHRONOUS :: recvbuf	33		
		YNCHRONOUS :: recvcounts(*), displs(*)	34		
	E(MPI_Comm), INTENT(35		
	E(MPI_Request), INTE EGER, OPTIONAL, INTE		36 37		
			38		
Fortran			39		
MPI_IALI		NDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, I, REQUEST, IERROR)	40		
<tvr< td=""><td><pre>> SENDBUF(*), RECV</pre></td><td></td><td>41</td></tvr<>	<pre>> SENDBUF(*), RECV</pre>		41		
• -		TYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	42		
	REQUEST, IERR	OR	43 44		
This	call starts a nonblockin	ng variant of MPI_ALLGATHERV (see Section 6.7).	44		
			47		

1 6.12.6 Nonblocking All-to-All Scatter/Gather $\mathbf{2}$ 3 4 MPI_IALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request) 56 IN sendbuf starting address of send buffer (choice) 7 IN 8 sendcount number of elements sent to each process (non-negative integer) 9 10 IN sendtype data type of send buffer elements (handle) 11 OUT recvbuf address of receive buffer (choice) 12IN recvcount number of elements received from any process 13 14(non-negative integer) 15IN recvtype data type of receive buffer elements (handle) 16communicator (handle) IN comm 17OUT communication request (handle) 18 request 19 20C binding 21int MPI_Ialltoall(const void *sendbuf, int sendcount, 22 MPI_Datatype sendtype, void *recvbuf, int recvcount, 23MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 24Fortran 2008 binding 25MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 26comm, request, ierror) 27TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 28INTEGER, INTENT(IN) :: sendcount, recvcount 29 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 30 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 31 TYPE(MPI_Comm), INTENT(IN) :: comm 32 TYPE(MPI_Request), INTENT(OUT) :: request 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 35 Fortran binding 36 MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 37 COMM, REQUEST, IERROR) 38 <type> SENDBUF(*), RECVBUF(*) 39 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 40 This call starts a nonblocking variant of MPI_ALLTOALL (see Section 6.8). 41 4243 44 45 46 47 48

MPI_IAL	LTOALLV(sendbuf, send recvtype, comm,	counts, sdispls, sendtype, recvbuf, recvcounts, rdispls, request)	$\frac{1}{2}$
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank	4 5 6 7
IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j	8 9 10
IN	sendtype	data type of send buffer elements (handle)	11
OUT	recvbuf	address of receive buffer (choice)	12 13
IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank	14 15 16
IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	17 18 19 20
IN	recvtype	data type of receive buffer elements (handle)	21
IN	comm	communicator (handle)	22
OUT	request	communication request (handle)	23 24
C bindi int MPI	_Ialltoallv(const vo const int sdi const int rec	<pre>Did *sendbuf, const int sendcounts[], spls[], MPI_Datatype sendtype, void *recvbuf, vcounts[], const int rdispls[], recvtype, MPI_Comm comm, MPI_Request *request)</pre>	25 26 27 28 29 30
MPI_Ial TYP INT TYP TYP TYP	rdispls, recv E(*), DIMENSION(), EGER, INTENT(IN), AS recvcounts(*) E(MPI_Datatype), INT	CENT(IN) :: sendtype, recvtype ASYNCHRONOUS :: recvbuf (IN) :: comm CNT(OUT) :: request	 31 32 33 34 35 36 37 38 39 40 41
MPI_IAL		IDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, TYPE, COMM, REQUEST, IERROR) /BUF(*)	42 43 44 45 46
INT		SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), M, REQUEST, IERROR	47 48

	240	CH	IAPTER 6. COLLECTIVE COMMUNICATION
1 2 3	This ca	all starts a nonblocking variar	nt of MPI_ALLTOALLV (see Section 6.8).
3 4 5	MPI_IALLT	OALLW(sendbuf, sendcounts, sendcounts, secvtypes, comm, request	sdispls, sendtypes, recvbuf, recvcounts, rdispls,)
6	IN	sendbuf	starting address of send buffer (choice)
7 8 9 10	IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each rank (array of non-negative integers)
11 12 13 14	IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers)
15 16 17	IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)
18 19	OUT	recvbuf	address of receive buffer (choice)
20 21 22	IN	recvcounts	integer array (of length group size) specifying the number of elements that can be received from each rank (array of non-negative integers)
23 24 25 26	IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)
27 28 29 30	IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)
31	IN	comm	communicator (handle)
32 33	Ουτ	request	communication request (handle)
34 35 36 37 38 39 40	C binding int MPI_I	alltoallw(const void *ser const int sdispls[], void *recvbuf, const	<pre>dbuf, const int sendcounts[], const MPI_Datatype sendtypes[], int recvcounts[], const int rdispls[], ecvtypes[], MPI_Comm comm,</pre>
41 42 43 44 45 46 47 48	MPI_Iallt TYPE(INTEG	recvcounts, rdispls, *), DIMENSION(), INTENT ER, INTENT(IN), ASYNCHRON recvcounts(*), rdisp	<pre>s, sdispls, sendtypes, recvbuf, recvtypes, comm, request, ierror) C(IN), ASYNCHRONOUS :: sendbuf HOUS :: sendcounts(*), sdispls(*), bls(*) , ASYNCHRONOUS :: sendtypes(*),</pre>

```
TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                         1
                                                                                         2
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
               RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
                                                                                         10
               RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR
                                                                                         11
    This call starts a nonblocking variant of MPI_ALLTOALLW (see Section 6.8).
                                                                                         12
                                                                                         13
                                                                                         14
6.12.7 Nonblocking Reduce
                                                                                         15
                                                                                         16
                                                                                         17
MPI_IREDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, request)
                                                                                         18
                                       address of send buffer (choice)
  IN
           sendbuf
                                                                                         19
  OUT
                                                                                         20
           recvbuf
                                       address of receive buffer (choice, significant only at
                                                                                        21
                                       root)
                                                                                         22
  IN
                                       number of elements in send buffer (non-negative
           count
                                                                                        23
                                       integer)
                                                                                         ^{24}
  IN
                                       data type of elements of send buffer (handle)
           datatype
                                                                                         25
                                                                                         26
  IN
                                       reduce operation (handle)
           op
                                                                                         27
  IN
                                       rank of root process (integer)
           root
                                                                                         28
  IN
                                       communicator (handle)
           comm
                                                                                         29
                                                                                         30
  OUT
                                       communication request (handle)
           request
                                                                                         31
                                                                                         32
C binding
                                                                                         33
int MPI_Ireduce(const void *sendbuf, void *recvbuf, int count,
                                                                                        34
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
                                                                                        35
               MPI_Request *request)
                                                                                         36
Fortran 2008 binding
                                                                                         37
MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
                                                                                         38
               ierror)
                                                                                         39
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                         40
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                         41
    INTEGER, INTENT(IN) :: count, root
                                                                                         42
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                         43
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                         44
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                         45
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                         46
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                         47
                                                                                         48
```

1	Fortran b	oinding		
2		0	CVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,	
3	IERROR)			
4	• -	<pre>> SENDBUF(*),</pre>		
5	INTEG	ER COUNT, DATA	TYPE, OP, ROOT, COMM, REQUEST, IERROR	
6 7	This c	call starts a nonbl	ocking variant of MPI_REDUCE (see Section 6.9.1).	
8	A davi	ce to implemento	rs. The implementation is explicitly allowed to use different	
9		-	g and nonblocking reduction operations that might change the	
10	-		the operations. However, as for MPI_REDUCE, it is strongly	
11 12	recor	nmended that MF	PI_IREDUCE be implemented so that the same result be obtained	
13			is applied on the same arguments, appearing in the same order.	
14			event optimizations that take advantage of the physical location advice to implementors.)	
15 16	_	· · ·		
17			operations which are not truly associative, the result delivered	
18	-	-	e nonblocking reduction may not exactly equal the result deliv-	
19		c. (End of advice	eduction, even when specifying the same arguments in the same to users.)	
20	oruei	. (Ena of addice		
21	6.12.8 N	onblocking All-R	educe	
22	0.12.0			
23 24				
24 25	MPI_IALLI	REDUCE(sendbuf,	recvbuf, count, datatype, op, comm, request)	
26	IN	sendbuf	starting address of send buffer (choice)	
27 28	OUT	recvbuf	starting address of receive buffer (choice)	
29 30	IN	count	number of elements in send buffer (non-negative integer)	
31	IN	datatype	data type of elements of send buffer (handle)	
32 33	IN	ор	operation (handle)	
34	IN	comm	communicator (handle)	
35	OUT	request	communication request (handle)	
36				
37	C binding			
38 39	int MPI_I		t void *sendbuf, void *recvbuf, int count,	
40		· · · · ·	rpe datatype, MPI_Op op, MPI_Comm comm,	
41		MP1_Reques	st *request)	
42	Fortran 2	2008 binding		
43	MPI_Iallr		recvbuf, count, datatype, op, comm, request,	
44		ierror)	· · · · · · · · · · · · · · · · · · ·	
45), INTENT(IN), ASYNCHRONOUS :: sendbuf	
46		ER, INTENT(IN)), ASYNCHRONOUS :: recvbuf	
47 48			INTENT(IN) :: datatype	
40	1111(

```
1
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                        2
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST,
               IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
                                                                                        10
                                                                                        11
    This call starts a nonblocking variant of MPI_ALLREDUCE (see Section 6.9.6).
                                                                                        12
                                                                                        13
       Nonblocking Reduce-Scatter with Equal Blocks
6.12.9
                                                                                        14
                                                                                        15
                                                                                        16
MPI_IREDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                        17
              request)
                                                                                        18
 IN
           sendbuf
                                       starting address of send buffer (choice)
                                                                                        19
                                                                                        20
 OUT
           recvbuf
                                       starting address of receive buffer (choice)
                                                                                        21
 IN
           recvcount
                                       element count per block (non-negative integer)
                                                                                        22
 IN
           datatype
                                       data type of elements of send and receive buffers
                                                                                        23
                                       (handle)
                                                                                        24
                                                                                        25
                                       operation (handle)
 IN
           ор
                                                                                        26
 IN
                                       communicator (handle)
           comm
                                                                                        27
 OUT
           request
                                       communication request (handle)
                                                                                        28
                                                                                        29
C binding
                                                                                        30
int MPI_Ireduce_scatter_block(const void *sendbuf, void *recvbuf,
                                                                                        31
               int recvcount, MPI_Datatype datatype, MPI_Op op,
                                                                                        32
              MPI_Comm comm, MPI_Request *request)
                                                                                        33
                                                                                        34
Fortran 2008 binding
                                                                                        35
MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                        36
              request, ierror)
                                                                                        37
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                        38
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                        39
    INTEGER, INTENT(IN) :: recvcount
                                                                                        40
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                        41
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                        42
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                        43
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                        44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        45
                                                                                        46
Fortran binding
                                                                                        47
MPI_IREDUCE_SCATTER_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
                                                                                        48
              REQUEST, IERROR)
```

```
244
                                         CHAPTER 6. COLLECTIVE COMMUNICATION
1
          <type> SENDBUF(*), RECVBUF(*)
\mathbf{2}
          INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR
3
         This call starts a nonblocking variant of MPI_REDUCE_SCATTER_BLOCK (see Sec-
4
     tion 6.10.1).
5
6
     6.12.10 Nonblocking Reduce-Scatter
7
8
9
     MPI_IREDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, request)
10
11
       IN
                 sendbuf
                                             starting address of send buffer (choice)
12
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
13
       IN
14
                 recvcounts
                                             non-negative integer array specifying the number of
15
                                             elements in result distributed to each process. This
16
                                             array must be identical on all calling processes.
17
       IN
                 datatype
                                             data type of elements of input buffer (handle)
18
       IN
                                             operation (handle)
                 ор
19
                                             communicator (handle)
20
       IN
                 comm
21
       OUT
                 request
                                             communication request (handle)
22
23
     C binding
24
     int MPI_Ireduce_scatter(const void *sendbuf, void *recvbuf,
25
                    const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
26
                    MPI_Comm comm, MPI_Request *request)
27
     Fortran 2008 binding
28
     MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
29
                    request, ierror)
30
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
31
32
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
33
          INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
34
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
          TYPE(MPI_Op), INTENT(IN) :: op
36
          TYPE(MPI_Comm), INTENT(IN) :: comm
37
          TYPE(MPI_Request), INTENT(OUT) :: request
38
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     Fortran binding
40
     MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
41
                    REQUEST, IERROR)
42
          <type> SENDBUF(*), RECVBUF(*)
43
          INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR
44
45
         This call starts a nonblocking variant of MPI_REDUCE_SCATTER (see Section 6.10.2).
46
47
48
```

6.12.11 Nonblocking Inclusive Scan

0.12.11	Nonbiocking melusive Scan		2
			3
MPI_ISC	AN(sendbuf, recvbuf, count, data	itype, op, comm, request)	4
IN	sendbuf	starting address of send buffer (choice)	5
OUT	recvbuf	starting address of receive buffer (choice)	6 7
		- · · · · · · · · · · · · · · · · · · ·	8
IN	count	number of elements in input buffer (non-negative integer)	9
IN	datatype		10
		data type of elements of input buffer (handle)	11
IN	ор	operation (handle)	12 13
IN	comm	communicator (handle)	13
OUT	request	communication request (handle)	15
~			16
C bindi	0	with surveying that survey	17
int MPI_		, void *recvbuf, int count, e, MPI_Op op, MPI_Comm comm,	18
	MPI_Request *request		19 20
		, ,	20
	2008 binding		22
		, datatype, op, comm, request, ierror)	23
	E(*), DIMENSION(), INTEND E(*), DIMENSION(), ASYNCH	((IN), ASYNCHRONOUS :: sendbuf	24
	EGER, INTENT(IN) :: count		25
	E(MPI_Datatype), INTENT(IN)) :: datatvpe	26
	E(MPI_Op), INTENT(IN) :: or		27
	E(MPI_Comm), INTENT(IN) ::		28
TYPE	E(MPI_Request), INTENT(OUT)	:: request	29
INTE	EGER, OPTIONAL, INTENT(OUT)	:: ierror	30 31
Fortran	binding		32
		, DATATYPE, OP, COMM, REQUEST, IERROR)	33
	<pre>SENDBUF(*), RECVBUF(*)</pre>		34
INTE	EGER COUNT, DATATYPE, OP, O	COMM, REQUEST, IERROR	35
This	call starts a nonblocking varia	nt of MPI_SCAN (see Section 6.11).	36
1 1115	can starts a honorocking varia		37
			38
			39
	Ŷ		40 41
			42
			43
			44
			45
			46
			47

1

```
246
                                          CHAPTER 6. COLLECTIVE COMMUNICATION
1
     6.12.12 Nonblocking Exclusive Scan
\mathbf{2}
3
4
     MPI_IEXSCAN(sendbuf, recvbuf, count, datatype, op, comm, request)
5
       IN
                 sendbuf
                                             starting address of send buffer (choice)
6
       OUT
7
                 recvbuf
                                              starting address of receive buffer (choice)
8
       IN
                                              number of elements in input buffer (non-negative
                 count
9
                                             integer)
10
                                              data type of elements of input buffer (handle)
       IN
                 datatype
11
       IN
                                              operation (handle)
12
                 op
13
       IN
                                              intracommunicator (handle)
                 comm
14
       OUT
                                              communication request (handle)
                 request
15
                                                                                 16
     C binding
17
     int MPI_Iexscan(const void *sendbuf, void *recvbuf, int count,
18
                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
19
                     MPI_Request *request)
20
21
     Fortran 2008 binding
22
     MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
23
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
^{24}
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
25
          INTEGER, INTENT(IN) :: count
26
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
          TYPE(MPI_Op), INTENT(IN) :: op
28
          TYPE(MPI_Comm), INTENT(IN) :: comm
29
          TYPE(MPI_Request), INTENT(OUT) :: request
30
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
     Fortran binding
32
     MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
33
          <type> SENDBUF(*), RECVBUF(*)
34
          INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
35
36
          This call starts a nonblocking variant of MPI_EXSCAN (see Section 6.11.2).
37
38
             Persistent Collective Operations
     6.13
39
40
     Many parallel computation algorithms involve repetitively executing a collective commu-
41
     nication operation with the same arguments each time. As with persistent point-to-point
42
     operations (see Section 3.9), persistent collective operations allow the MPI programmer to
43
     specify operations that will be reused frequently (with fixed arguments). MPI can be de-
44
     signed to select a more efficient way to perform the collective operation based on the param-
45
     eters specified when the operation is initialized. This "planned-transfer" approach [52, 41]
46
     can offer significant performance benefits for programs with repetitive communication pat-
```

47

48

terns.

In terms of data movement, each persistent collective operation has the same effect as its blocking and nonblocking counterparts for intracommunicators and intercommunicators after completion. Likewise, upon completion, persistent collective reduction operations perform the same operation as their blocking and nonblocking counterparts, and the same restrictions and recommendations on reduction orders apply (see also Section 6.9.1).

Initialization calls for MPI persistent collective operations are non-local and follow all the existing rules for collective operations, in particular ordering; programs that do not conform to these restrictions are erroneous. After initialization, all arrays associated with input arguments (such as arrays of counts, displacements, and datatypes in the vector versions of the collectives) must not be modified until the corresponding persistent request is freed with MPI_REQUEST_FREE.

According to the definitions in Section 2.4.2, the persistent collective initialization procedures are incomplete. They are also non-local procedures because they may or may not return before they are called in all MPI processes of the process group associated with the specified communicator.

Advice to users. This is one of the exceptions in which incomplete procedures are non-local and therefore blocking. (End of advice to users.)

The request argument is an output argument that can be used zero or more times with MPI_START or MPI_STARTALL in order to start the collective operation. The request is initially inactive after the initialization call. Once initialized, persistent collective operations can be started in any order and the order can differ among processes in the communicator.

Rationale. All ordering requirements that an implementation may need to match up collective operations across the communicator are achieved through the ordering requirements of the initialization functions. This enables out-of-order starts for the persistent operations, and particularly supports their use in MPI_STARTALL. (*End of rationale.*)

Advice to implementors. An MPI implementation should do no worse than duplicating the communicator during the initialization function, caching the input arguments, and calling the appropriate nonblocking collective function, using the cached arguments, during MPI_START. High-quality implementations should be able to amortize setup costs and further optimize by taking advantage of early-binding, such as efficient and effective pre-allocation of certain resources and algorithm selection. (*End* of advice to implementors.)

A request must be inactive when it is started. Starting the operation makes the request active. Once any process starts a persistent collective operation, it must complete that operation and all other processes in the communicator must eventually start (and complete) the same persistent collective operation. Persistent collective operations cannot be matched with blocking or nonblocking collective operations. Completion of a persistent collective operation makes the corresponding request inactive. After starting a persistent collective operation, all associated send buffers must not be modified and all associated receive buffers must not be accessed until the corresponding persistent request is completed.

Completing a persistent collective request, for example using MPI_TEST or 46 MPI_WAIT, makes it inactive, but does not free the request. This is the same behavior as 47 for persistent point-to-point requests. Inactive persistent collective requests can be freed 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6 7

8 9

10

11

12

13

14

15

16

17

18 19

20

21

22

23 24

25

26

27

28

29 30

 31

32

33

34

35

36

37

38

39

40 41

42

43

44

```
1
      using MPI_REQUEST_FREE. It is erroneous to free an active persistent collective request.
\mathbf{2}
      Persistent collective operations cannot be canceled; it is erroneous to use MPI_CANCEL on
3
      a persistent collective request.
4
          For every nonblocking collective communication operation in MPI, there is a corre-
5
      sponding persistent collective operation with the analogous API signature.
6
          The collective persistent API signatures include an MPI_INFO object in order to support
\overline{7}
     optimization hints and other information that may be non-standard. Persistent collective
8
      operations may be optimized during communicator creation or by the initialization opera-
9
      tion of an individual persistent collective. Note that communicator-scoped hints should be
10
      provided using MPI_COMM_SET_INFO while, for operation-scoped hints, they are supplied
11
     to the persistent collective communication initialization functions using the info argument.
12
13
     6.13.1
             Persistent Barrier Synchronization
14
15
16
      MPI_BARRIER_INIT(comm, info, request)
17
       IN
                  comm
                                               communicator (handle)
18
19
       IN
                  info
                                               info argument (handle)
20
        OUT
                  request
                                               communication request (handle)
21
22
     C binding
23
      int MPI_Barrier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request)
24
25
      Fortran 2008 binding
26
      MPI_Barrier_init(comm, info, request, ierror)
27
          TYPE(MPI_Comm), INTENT(IN) :: comm
28
          TYPE(MPI_Info), INTENT(IN) :: info
29
          TYPE(MPI_Request), INTENT(OUT) :: request
30
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
      Fortran binding
32
      MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR)
33
          INTEGER COMM, INFO, REQUEST, IERROR
34
35
          Creates a persistent collective communication request for the barrier operation.
36
37
38
39
40
41
42
43
44
45
46
47
48
```

6.13.2 Persistent Broadcast

0.13.2 F	ersistent Dioaucast		2
			3
MPI_BCAS	ST_INIT(buffer, count, datatyp	e, root, comm, info, request)	4 5
INOUT	buffer	starting address of buffer (choice)	6
IN	count	number of entries in buffer (non-negative integer)	7
IN	datatype	data type of buffer (handle)	8 9
IN	root	rank of broadcast root (integer)	9 10
IN	comm	communicator (handle)	11
IN	info	info argument (handle)	12
OUT	request	communication request (handle)	13 14
			15
C binding			16
int MPI_E		<pre>int count, MPI_Datatype datatype, comm, MPI_Info info, MPI_Request *request)</pre>	17
		.omm, MFI_INIO INIO, MFI_Request *request)	18 19
	2008 binding	stung most com info memorat isomer)	20
	(*), DIMENSION(), ASYNC	atype, root, comm, info, request, ierror) HRONOUS :: buffer	21
	ER, INTENT(IN) :: count,		22
	MPI_Datatype), INTENT(IN		23 24
	(MPI_Comm), INTENT(IN) ::		25
	<pre>[MPI_Info), INTENT(IN) :: [MPI_Request), INTENT(OUT</pre>		26
	ER, OPTIONAL, INTENT(OUT	-	27
Fortran b	pinding		28 29
	3	ATYPE, ROOT, COMM, INFO, REQUEST, IERROR)	30
	> BUFFER(*)		31
INTEG	ER COUNT, DATATYPE, ROOT	, COMM, INFO, REQUEST, IERROR	32
Create	es a persistent collective com	munication request for the broadcast operation.	33 34
			35
			36
			37
			38 39
			40
			41
			42
			43
			$44 \\ 45$
			46
			47
			48

6.13.3 Persistent Gather 1 $\mathbf{2}$ 3 4 MPI_GATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, 5info, request) 6 IN sendbuf starting address of send buffer (choice) 7 8 IN sendcount number of elements in send buffer (non-negative 9 integer) 10 IN sendtype data type of send buffer elements (handle) 11 OUT recvbuf address of receive buffer (choice, significant only at 12root) 13 IN number of elements for any single receive 14recvcount (non-negative integer, significant only at root) 1516IN data type of recv buffer elements (handle, significant recvtype 17only at root) 18 IN rank of receiving process (integer) root 19 IN communicator (handle) comm 2021IN info info argument (handle) 22 OUT request communication request (handle) 23 24C binding 25int MPI_Gather_init(const void *sendbuf, int sendcount, 26MPI_Datatype sendtype, void *recvbuf, int recvcount, 27MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info, 28MPI_Request *request) 29 30 Fortran 2008 binding 31 MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 32 root, comm, info, request, ierror) 33 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 34 INTEGER, INTENT(IN) :: sendcount, recvcount, root 35 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 36 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 37 TYPE(MPI_Comm), INTENT(IN) :: comm 38 TYPE(MPI_Info), INTENT(IN) :: info 39 TYPE(MPI_Request), INTENT(OUT) :: request 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 Fortran binding 42MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 43 ROOT, COMM, INFO, REQUEST, IERROR) 44 <type> SENDBUF(*), RECVBUF(*) 45 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, 46 REQUEST, IERROR 47 48

Creates a persistent collective communication request for the gather operation.			
MPI_GATH	IERV_INIT(sendbuf, sendcount root, comm, info, request	, sendtype, recvbuf, recvcounts, displs, recvtype,)	3 4 5
IN	sendbuf	starting address of send buffer (choice)	6
IN	sendcount	number of elements in send buffer (non-negative integer)	7 8 9
IN	sendtype	data type of send buffer elements (handle)	10
OUT	recvbuf	address of receive buffer (choice, significant only at root)	11 12
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)	13 14 15 16
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)	17 18 19 20
IN	recvtype	data type of recv buffer elements (handle, significant only at root)	21 22
IN	root	rank of receiving process (integer)	23
IN	comm	communicator (handle)	24 25
IN	info	info argument (handle)	26
OUT	request	communication request (handle)	27
			28
C binding	g A		29 30
int MPI_G	atherv_init(const void *s		31
		e, void *recvbuf, const int recvcounts[],	32
		<pre>MPI_Datatype recvtype, int root, nfo info, MPI_Request *request)</pre>	33
		iio inio, in i_nequest wiequest)	34 35
	008 binding	t condturne nearbuf nearcounts displa	36
MP1_Gatne		t, sendtype, recvbuf, recvcounts, displs, , info, request, ierror)	37
TYPE((IN), ASYNCHRONOUS :: sendbuf	38
INTEG	ER, INTENT(IN) :: sendcou	nt, root	39
	MPI_Datatype), INTENT(IN)		40
	*), DIMENSION(), ASYNCH		41 42
	MPI_Comm), INTENT(IN) ::	<pre>IOUS :: recvcounts(*), displs(*) comm</pre>	43
	<pre>MPI_Info), INTENT(IN) ::</pre>		44
TYPE(MPI_Request), INTENT(OUT)	:: request	45
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	46
Fortran b	inding		47 48

	202	CI	TAFTER 0. COLLECTIVE COMMONICATION
1 2 3 4 5	<type< th=""><th>RECVTYPE, ROOT, COMM >> SENDBUF(*), RECVBUF(*)</th><th>NT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, , INFO, REQUEST, IERROR) RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, , IERROR</th></type<>	RECVTYPE, ROOT, COMM >> SENDBUF(*), RECVBUF(*)	NT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, , INFO, REQUEST, IERROR) RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, , IERROR
6 7	Create	es a persistent collective comm	nunication request for the gathery operation.
8 9 10 11	6.13.4 Pe	ersistent Scatter	
12 13	MPI_SCAT	TER_INIT(sendbuf, sendcount comm, info, request)	, sendtype, recvbuf, recvcount, recvtype, root,
14 15	IN	sendbuf	address of send buffer (choice, significant only at root)
16 17 18	IN	sendcount	number of elements sent to each process (non-negative integer, significant only at root)
19 20	IN	sendtype	data type of send buffer elements (handle, significant only at root)
21	OUT	recvbuf	address of receive buffer (choice)
22 23 24	IN	recvcount	number of elements in receive buffer (non-negative integer)
25	IN	recvtype	data type of receive buffer elements (handle)
26 27	IN	root	rank of sending process (integer)
27	IN	comm	communicator (handle)
29	IN	info	info argument (handle)
30 31	OUT	request	communication request (handle)
32 33 34 35 36 37	C binding int MPI_S	catter_init(const void *: MPI_Datatype sendtyp	e, void *recvbuf, int recvcount, e, int root, MPI_Comm comm, MPI_Info info,
38		2008 binding	
39 40	MPI_Scatt		nt, sendtype, recvbuf, recvcount,
40	TYPE(• •	, info, request, ierror) Γ(IN), ASYNCHRONOUS :: sendbuf
42		ER, INTENT(IN) :: sendcou	
43	TYPE(MPI_Datatype), INTENT(IN)) :: sendtype, recvtype
44		(*), DIMENSION(), ASYNCI	
45 46		<pre>(MPI_Comm), INTENT(IN) :: (MPI_Info), INTENT(IN) ::</pre>	
47		[MPI_Request), INTENT(OUT)	
48		ER, OPTIONAL, INTENT(OUT)	-

	binding		1
MPI_SCA		SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	2 3
/+		T, COMM, INFO, REQUEST, IERROR)	4
•	pe> SENDBUF(*), REC	DTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,	5
	REQUEST, IER		6
G			7
Cre	ates a persistent collect	ive communication request for the scatter operation.	8
			9
MPI_SC	ATTERV_INIT(sendbuf,	sendcounts, displs, sendtype, recvbuf, recvcount, recvtype,	10 11
	root, comm, info	o, request)	11
IN	sendbuf	address of send buffer (choice, significant only at	13
		root)	14
IN	sendcounts	non-negative integer array (of length group size)	15
		specifying the number of elements to send to each	16
		rank (significant only at root)	17
IN	displs	integer array (of length group size). Entry i specifies	18 19
		the displacement (relative to sendbuf) from which to	20
		take the outgoing data to process i (significant only	21
		at root)	22
IN	sendtype	data type of send buffer elements (handle, significant only at root)	23
0.U.T			24
OUT	recvbuf	address of receive buffer (choice, significant only at root)	25 26
IN	recvcount	number of elements in receive buffer (non-negative	27
		integer)	28
IN	recvtype	data type of receive buffer elements (handle)	29 30
IN	root	rank of sending process (integer)	31
IN	comm	communicator (handle)	32
IN	info	info argument (handle)	33
OUT	request	communication request (handle)	34 35
001	request	communication request (nandle)	36
C bindi	ing		37
		t void *sendbuf, const int sendcounts[],	38
		<pre>pls[], MPI_Datatype sendtype, void *recvbuf,</pre>	39
		, MPI_Datatype recvtype, int root, MPI_Comm comm,	40
	MPI_Info info	, MPI_Request *request)	41 42
Fortran	2008 binding		42
MPI_Sca	tterv_init(sendbuf,	sendcounts, displs, sendtype, recvbuf,	44
		cvtype, root, comm, info, request, ierror)	45
		, INTENT(IN), ASYNCHRONOUS :: sendbuf	46
		SYNCHRONOUS :: sendcounts(*), displs(*) TENT(IN) :: sendtype, recvtype	47
115	- Curt-Davacybe, IN	TTUILING BETTOPHE, TECKONHE	48

1 2 3 4 5 6 7 8 9 10 11 12 13	INTH TYPH TYPH INTH Fortran MPI_SCAT	RECVCOUNT, De> SENDBUF(*), EGER SENDCOUNTS(<pre>:: recvcount, ENT(IN) :: com ENT(IN) :: inf ENTENT(OUT) :: ENTENT(OUT) :: UF, SENDCOUNTS RECVTYPE, ROO ECVBUF(*)</pre>	, root mm fo : request : ierror S, DISPLS, SENDTYPE, RECVBUF, OT, COMM, INFO, REQUEST, IERROR) , SENDTYPE, RECVCOUNT, RECVTYPE, R	QOOT,
14	Crea	tes a persistent co	ective communi	ication request for the scattery operation	n.
15 16 17 18 19 20	6.13.5	Persistent Gather-1	o-all Ibuf, sendcount,	sendtype, recvbuf, recvcount, recvtype, co	
21 22	IN	sendbuf		arting address of send buffer (choice)	
23 24	IN	sendcount	nu	umber of elements in send buffer (non-negatiteger)	ive
25	IN	sendtype		ta type of send buffer elements (handle)	
26 27	OUT	recvbuf		dress of receive buffer (choice)	
28 29	IN	recvcount		mber of elements received from any process on-negative integer)	ł
30 31	IN	recvtype	da	ta type of receive buffer elements (handle)	
32	IN	comm	COL	mmunicator (handle)	
33	IN	info	inf	fo argument (handle)	
34 35	OUT	request	COL	mmunication request (handle)	
36 37 38 39 40 41	C bindin int MPI	Allgather_init(MPI_Dataty MPI_Dataty	pe sendtype, v	endbuf, int sendcount, void *recvbuf, int recvcount, MPI_Comm comm, MPI_Info info,	
42 43 44 45 46 47 48	MPI_Allg TYPE INTE TYPE	recvtype, E(*), DIMENSION(EGER, INTENT(IN)	comm, info, re .), INTENT(IN :: sendcount, INTENT(IN) ::	sendtype, recvtype	

ΤY	YPE(MPI_Comm), INTEN	NT(IN) :: comm ¹
ΤY	<pre>/PE(MPI_Info), INTEN</pre>	NT(IN) :: info 2
	-	NTENT(OUT) :: request ³
II	NTEGER, OPTIONAL, IN	ITENT(OUT) :: ierror 4
Fortra	an binding	5
	•	JF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 7
	RECVTYPE, C	OMM, INFO, REQUEST, IERROR)
	type> SENDBUF(*), RE	9
II	NTEGER SENDCOUNT, SE IERROR	ENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
Cı	reates a persistent colle	ective communication request for the allgather operation.
MPI_A	LLGATHERV_INIT(sen comm, info, re	dbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, equest)
IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice) 22
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process
IN	displs	integer array (of length group size). Entry i specifies 22 the displacement (relative to recvbuf) at which to 22 place the incoming data from process i 22
IN	recvtype	data type of receive buffer elements (handle) ³⁰
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle) ³⁴
		38
C bin		30
int MF	-	const void *sendbuf, int sendcount,
		e sendtype, void *recvbuf, const int recvcounts[],
		fo, MPI_Request *request)
D (41
	an 2008 binding	ouf, sendcount, sendtype, recvbuf, recvcounts, 44
III I_AJ	-	buf, sendcount, sendtype, recvbuf, recvcounts, 43 vtype, comm, info, request, ierror) 44
ΤY	-	.), INTENT(IN), ASYNCHRONOUS :: sendbuf
	NTEGER, INTENT(IN)	
		INTENT(IN) :: sendtype, recvtype 47
ΤY	<pre>/PE(*), DIMENSION(</pre>	A), ASYNCHRONOUS :: recvbuf

1 2 3 4 5 6 7 8 9 10	TYPE(TYPE(TYPE(INTEG Fortran h MPI_ALLGA	<pre>(MPI_Comm), INTENT(IN) (MPI_Info), INTENT(IN) (MPI_Request), INTENT(ON ER, OPTIONAL, INTENT(ON Dinding ATHERV_INIT(SENDBUF, SEN DISPLS, RECVTYPE, >> SENDBUF(*), RECVBUF(: ER SENDCOUNT, SENDTYPE</pre>	<pre>:: info UT) :: request UT) :: ierror NDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, COMM, INFO, REQUEST, IERROR) *) , RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,</pre>
12 13	Create	INFO, REQUEST, IE	mmunication request for the allgathery operation.
14 15 16 17 18 19	6.13.6 P	ersistent All-to-All Scatter	
20	IN	sendbuf	starting address of send buffer (choice)
21 22	IN	sendcount	number of elements sent to each process
23			(non-negative integer)
24 25	IN	sendtype	data type of send buffer elements (handle)
26	OUT	recvbuf	address of receive buffer (choice)
27 28	IN	recvcount	number of elements received from any process (non-negative integer)
29 30	IN	recvtype	data type of receive buffer elements (handle)
31	IN	comm	communicator (handle)
32	IN	info	info argument (handle)
33 34	OUT	request	communication request (handle)
35 36 37 38 39 40	C binding int MPI_A	Alltoall_init(const void MPI_Datatype sendt	d *sendbuf, int sendcount, ype, void *recvbuf, int recvcount, ype, MPI_Comm comm, MPI_Info info, st)
41 42 43 44 45 46 47 48	MPI_Allto TYPE(INTEC TYPE(TYPE(recvtype, comm, in (*), DIMENSION(), INT ER, INTENT(IN) :: sende	IN) :: sendtype, recvtype NCHRONOUS :: recvbuf

TYPE INTE Fortran MPI_ALLT <typ INTE</typ 	DALL_INIT(SENDBUF, SENDCO RECVTYPE, COMM, INFO > SENDBUF(*), RECVBUF(*) GER SENDCOUNT, SENDTYPE, IERROR) :: request) :: ierror UNT, SENDTYPE, RECVBUF, RECVCOUNT,	1 2 3 4 5 6 7 8 9 10 11 12 13
MPI_ALL	FOALLV_INIT(sendbuf, sendcon recvtype, comm, info, re	unts, sdispls, sendtype, recvbuf, recvcounts, rdispls, quest)	14 15
IN	sendbuf	starting address of send buffer (choice)	16
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank	17 18 19
IN	sdispls	Integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j	20 21 22 23
IN	sendtype	data type of send buffer elements (handle)	24
OUT	recvbuf	address of receive buffer (choice)	25
IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank	26 27 28 29
IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	30 31 32
IN	recvtype	data type of receive buffer elements (handle)	33
IN	comm	communicator (handle)	34 35
IN	info	info argument (handle)	36
OUT	request	communication request (handle)	37 38
C bindin	g		$\frac{39}{40}$
int MPI_		*sendbuf, const int sendcounts[],	40
	-	MPI_Datatype sendtype, void *recvbuf,	42
		s[], const int rdispls[], pe, MPI_Comm comm, MPI_Info info,	43
	MPI_Request *request		44
Dest		-	45 46
	2008 binding	ounts sdignly condition recubil	40 47
<pre>MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, info, request, ierror)</pre>			

```
1
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
\mathbf{2}
          INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
3
                     recvcounts(*), rdispls(*)
4
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
5
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
6
          TYPE(MPI_Comm), INTENT(IN) :: comm
7
          TYPE(MPI_Info), INTENT(IN) :: info
8
          TYPE(MPI_Request), INTENT(OUT) :: request
9
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
     Fortran binding
11
     MPI_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
12
                    RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
13
          <type> SENDBUF(*), RECVBUF(*)
14
          INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
15
                     RECVTYPE, COMM, INFO, REQUEST, IERROR
16
17
          Creates a persistent collective communication request for the alltoally operation.
18
19
20
21
22
23
^{24}
25
26
27
28
29
30
^{31}
32
33
34
35
36
37
38
39
40
^{41}
42
43
44
45
46
47
48
```

MPI_ALLTOALLW_INIT(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, info, request)			
IN	sendbuf	starting address of send buffer (choice)	3 4
IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each rank (array of non-negative integers)	5 6 7
IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers)	8 9 10 11
IN	sendtypes	Array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)	12 13 14 15
OUT	recvbuf	address of receive buffer (choice)	16
IN	recvcounts	integer array (of length group size) specifying the number of elements that can be received from each rank (array of non-negative integers)	17 18 19
IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)	20 21 22 23 24
IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)	25 26 27
IN	comm	communicator (handle)	28
IN	info	info argument (handle)	29 30
OUT	request	communication request (handle)	31
C binding			32 33
		<pre>*sendbuf, const int sendcounts[],</pre>	34
	<pre>const int sdispls[], void *recvbuf, const</pre>	<pre>const MPI_Datatype sendtypes[], int recvcounts[], const int rdispls[], ecvtypes[], MPI_Comm comm, MPI_Info info,</pre>	35 36 37

```
Fortran 2008 binding
```

MPI_Request *request)

```
MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                 41
             recvcounts, rdispls, recvtypes, comm, info, request, ierror)
                                                                                 42
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 43
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                 44
              recvcounts(*), rdispls(*)
                                                                                 45
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                 46
              recvtypes(*)
                                                                                 47
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 48
```

38 39

1 2 3 4 5	TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
6 7 8 9 10 11	<pre>Fortran binding MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,</pre>				
12 13	Crea	ates a persistent collec	ctive communication request for the alltoally operation.		
14 15 16	6.13.7	Persistent Reduce			
17	MPI_REI	DUCE_INIT(sendbuf, r	ecvbuf, count, datatype, op, root, comm, info, request)		
18 19	IN	sendbuf	address of send buffer (choice)		
20 21	OUT	recvbuf	address of receive buffer (choice, significant only at root)		
22 23 24	IN	count	number of elements in send buffer (non-negative integer)		
25	IN	datatype	data type of elements of send buffer (handle)		
26	IN	ор	reduce operation (handle)		
27 28	IN	root	rank of root process (integer)		
29	IN	comm	communicator (handle)		
30	IN	info	info argument (handle)		
31 32	OUT	request	communication request (handle)		
33 34 35 36 37	C bindi int MPI	_Reduce_init(const MPI_Datatype	<pre>void *sendbuf, void *recvbuf, int count, e datatype, MPI_Op op, int root, MPI_Comm comm, fo, MPI_Request *request)</pre>		
38	Fortran	2008 binding			
$\frac{39}{40}$	MPI_Red		recvbuf, count, datatype, op, root, comm, info,		
41	TYP	request, ier E(*). DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf		
42), ASYNCHRONOUS :: recvbuf		
43		EGER, INTENT(IN) :			
44 45	TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Op), INTENT(IN) :: op				
46		E(MPI_OP), INTENI(E(MPI_Comm), INTEN	-		
47		E(MPI_Info), INTEN			
48	TYP	E(MPI_Request), IN	TENT(OUT) :: request		

INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
Fortran binding MPI_REDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO,				
	REQUEST, IE		4 5	
	<type> SENDBUF(*), R</type>		6	
	INTEGER COUNT, DATAT	YPE, OP, ROOT, COMM, INFO, REQUEST, IERROR	7	
(Creates a persistent colle	ective communication request for the reduce operation.	8 9	
6 1 2	8 Persistent All-Reduc		10	
6.13.	o Persistent All-Reduc	Le Contraction of the second se	11	
			12 13	
MPI_	ALLREDUCE_INIT(send	buf, recvbuf, count, datatype, op, comm, info, request)	14	
IN	sendbuf	starting address of send buffer (choice)	15	
OU	T recvbuf	starting address of receive buffer (choice)	16	
IN	count	number of elements in send buffer (non-negative	17	
		integer)	18 19	
IN	datatype	data type of elements of send buffer (handle)	20	
IN	ор	operation (handle)	21	
IN	comm	communicator (handle)	22	
IN	info	info argument (handle)	23 24	
OU		communication request (handle)	25	
00	i lequest	communication request (nandie)	26	
C bi	nding		27	
	ě	onst void *sendbuf, void *recvbuf, int count,	28	
		e datatype, MPI_Op op, MPI_Comm comm,	29 30	
	MPI_Info in	fo, MPI_Request *request)	31	
Fort	ran 2008 binding		32	
MPI_		uf, recvbuf, count, datatype, op, comm, info,	33	
	request, ie		34	
		.), INTENT(IN), ASYNCHRONOUS :: sendbuf .), ASYNCHRONOUS :: recvbuf	35 36	
	INTEGER, INTENT(IN)		37	
	<pre>TYPE(MPI_Datatype),</pre>	INTENT(IN) :: datatype	38	
	<pre>TYPE(MPI_Op), INTENT</pre>	•	39	
	TYPE(MPI_Comm), INTE		40	
	TYPE(MPI_Info), INTE	NI(IN) :: 1nio NTENT(OUT) :: request	41 42	
	-	NTENT(OUT) :: ierror	43	
			44	
	r an binding ALLREDUCE INIT(SENDB	UF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO,	45	
	REQUEST, IE		46	
	<type> SENDBUF(*), R</type>	ECVBUF(*)	47 48	

```
262
                                         CHAPTER 6. COLLECTIVE COMMUNICATION
1
          INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
2
         Creates a persistent collective communication request for the all reduce operation.
3
4
     6.13.9 Persistent Reduce-Scatter with Equal Blocks
5
6
7
     MPI_REDUCE_SCATTER_BLOCK_INIT(sendbuf, recvbuf, recvcount, datatype, op, comm,
8
9
                    info, request)
10
                 sendbuf
       IN
                                            starting address of send buffer (choice)
11
                                            starting address of receive buffer (choice)
       OUT
                 recvbuf
12
       IN
                                            element count per block (non-negative integer)
13
                 recvcount
14
                                             data type of elements of send and receive buffers
       IN
                 datatype
15
                                             (handle)
16
       IN
                                             operation (handle)
                 ор
17
18
       IN
                                            communicator (handle)
                 comm
19
       IN
                 info
                                            info argument (handle)
20
       OUT
                 request
                                             communication request (handle)
21
22
     C binding
23
     int MPI_Reduce_scatter_block_init(const void *sendbuf, void *recvbuf,
^{24}
                    int recvcount, MPI_Datatype datatype, MPI_Op op,
25
                    MPI_Comm comm, MPI_Info info, MPI_Request *request)
26
27
     Fortran 2008 binding
28
     MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
29
                    comm, info, request, ierror)
30
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
^{31}
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
32
          INTEGER, INTENT(IN) :: recvcount
33
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
          TYPE(MPI_Op), INTENT(IN) :: op
35
          TYPE(MPI_Comm), INTENT(IN) :: comm
36
          TYPE(MPI_Info), INTENT(IN) :: info
37
          TYPE(MPI_Request), INTENT(OUT) :: request
38
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     Fortran binding
40
     MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP,
41
                    COMM, INFO, REQUEST, IERROR)
42
          <type> SENDBUF(*), RECVBUF(*)
43
          INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
44
45
          Creates a persistent collective communication request for the reduce-scatter with equal
46
     blocks operation.
47
```

6.13.10 Persistent Reduce-Scatter

MPI_REDUCE_SCATTER_INIT(sendbuf, recvbuf, recvcounts, datatype, op, comm, info,

_	request)		5
	• /		6
IN	sendbuf	starting address of send buffer (choice)	7
OUT	recvbuf	starting address of receive buffer (choice)	8
IN	recvcounts	non-negative integer array specifying the number of	9
		elements in result distributed to each process. This	10
		array must be identical on all calling processes.	11
IN	datatype	data type of elements of input buffer (handle)	12
	5.		13
IN	ор	operation (handle)	14
IN	comm	communicator (handle)	15
IN	info	info argument (handle)	16
		o (()	17
OUT	request	communication request (handle)	18
			19

C binding

<pre>int MPI_Reduce_scatter_init(const void *sendbuf, void *recvbuf,</pre>
<pre>const int recvcounts[], MPI_Datatype datatype, MPI_Op op,</pre>
MPI_Comm comm, MPI_Info info, MPI_Request *request)

Fortran 2008 binding MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm, info, request, ierror) TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*) TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding

MPI_REDUCE_SCATTER_INIT(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, INFO, REQUEST, IERROR

Creates a persistent collective communication request for the reduce-scatter operation.

202122

2324

25

26

27

28

29

30

31

32

33

34

35

36 37

38

39

40

41

42

```
CHAPTER 6. COLLECTIVE COMMUNICATION
     264
1
     6.13.11 Persistent Inclusive Scan
\mathbf{2}
3
4
     MPI_SCAN_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)
5
       IN
                 sendbuf
                                             starting address of send buffer (choice)
6
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
7
8
       IN
                                             number of elements in input buffer (non-negative
                 count
9
                                             integer)
10
       IN
                                             data type of elements of input buffer (handle)
                 datatype
11
       IN
                                             operation (handle)
12
                 ор
13
       IN
                 comm
                                             communicator (handle)
14
       IN
                 info
                                             info argument (handle)
15
16
       OUT
                 request
                                             communication request (handle)
17
18
     C binding
19
     int MPI_Scan_init(const void *sendbuf, void *recvbuf, int count,
20
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
21
                    MPI_Info info, MPI_Request *request)
22
     Fortran 2008 binding
23
     MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
24
                     ierror)
25
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
26
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
27
          INTEGER, INTENT(IN) :: count
28
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
          TYPE(MPI_Op), INTENT(IN) :: op
30
          TYPE(MPI_Comm), INTENT(IN) :: comm
^{31}
          TYPE(MPI_Info), INTENT(IN) :: info
32
          TYPE(MPI_Request), INTENT(OUT) :: request
33
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     Fortran binding
36
     MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
37
                    IERROR)
38
          <type> SENDBUF(*), RECVBUF(*)
39
          INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
40
          Creates a persistent collective communication request for the inclusive scan operation.
41
42
43
44
45
46
47
48
```

6.13.12 Persistent Exclusive Scan

0.15.	12 Persistent Exclusive 3	SCall	2
			3
MPI_	EXSCAN_INIT(sendbuf, ree	cvbuf, count, datatype, op, comm, info, request)	4
IN	sendbuf	starting address of send buffer (choice)	5 6
OU	T recvbuf	starting address of receive buffer (choice)	7
IN	count	number of elements in input buffer (non-negative	8
		integer)	9 10
IN	datatype	data type of elements of input buffer (handle)	10
IN	ор	operation (handle)	12
IN	comm	intracommunicator (handle)	13
IN	info	info argument (handle)	14 15
OU	T request	communication request (handle)	16
	,		17
C bi	nding		18
int 1		void *sendbuf, void *recvbuf, int count,	19
	• =	datatype, MPI_Op op, MPI_Comm comm,	20 21
	MP1_Info info	o, MPI_Request *request)	22
Fortran 2008 binding			23
MPI_		ecvbuf, count, datatype, op, comm, info, request,	24
<pre>ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf</pre>			25
TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf			26 27
	INTEGER, INTENT(IN) ::		27
	TYPE(MPI_Datatype), IN	TENT(IN) :: datatype	29
	TYPE(MPI_Op), INTENT(I		30
	TYPE(MPI_Comm), INTENT		31
	TYPE(MPI_Info), INTENT		32
	TYPE(MPI_Request), INT		33
	INTEGER, OPTIONAL, INT	ENI(UUI) :: lerror	34
Fort	ran binding		35
MPI_		ECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,	36
	IERROR)		37 38
	<type> SENDBUF(*), REC</type>		39
	INTEGER COUNT, DATATYP	E, OP, COMM, INFO, REQUEST, IERROR	40
(Creates a persistent collect	ive communication request for the exclusive scan operation.	41
			42
6.14	Correctness		43
			44

A correct, portable program must invoke collective communications so that deadlock will not deadlock will not occur, whether collective communications are synchronizing or not. The following examples definition definition of the following examples definition defin

Unofficial Draft for Comment Only

```
1
     Example 6.25 The following is erroneous.
\mathbf{2}
3
      switch(rank) {
          case 0:
4
               MPI_Bcast(buf1, count, type, 0, comm);
5
6
               MPI_Bcast(buf2, count, type, 1, comm);
               break;
7
8
          case 1:
               MPI_Bcast(buf2, count, type, 1, comm);
9
               MPI_Bcast(buf1, count, type, 0, comm);
10
11
               break;
     }
12
13
          We assume that the group of comm is \{0,1\}. Two processes execute two broadcast
14
      operations in reverse order. If the operation is synchronizing then a deadlock will occur.
15
          Collective operations must be executed in the same order at all members of the com-
16
      munication group.
17
18
     Example 6.26 The following is erroneous.
19
20
      switch(rank) {
21
          case 0:
22
               MPI_Bcast(buf1, count, type, 0, comm0);
23
               MPI_Bcast(buf2, count, type, 2, comm2);
^{24}
               break;
25
          case 1:
26
               MPI_Bcast(buf1, count, type, 1, comm1);
27
               MPI_Bcast(buf2, count, type, 0, comm0);
28
               break;
29
          case 2:
30
               MPI_Bcast(buf1, count, type, 2, comm2);
^{31}
               MPI_Bcast(buf2, count, type, 1, comm1);
32
               break;
33
      }
34
35
          Assume that the group of comm0 is \{0,1\}, of comm1 is \{1, 2\} and of comm2 is \{2,0\}. If
36
      the broadcast is a synchronizing operation, then there is a cyclic dependency: the broadcast
37
      in comm2 completes only after the broadcast in comm0; the broadcast in comm0 completes
38
      only after the broadcast in comm1; and the broadcast in comm1 completes only after the
39
      broadcast in comm2. Thus, the code will deadlock.
40
          Collective operations must be executed in an order so that no cyclic dependencies occur.
41
      Nonblocking collective operations can alleviate this issue.
42
43
     Example 6.27 The following is erroneous.
44
45
```

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, 0, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

Process zero executes a broadcast, followed by a blocking send operation. Process one first executes a blocking receive that matches the send, followed by broadcast call that matches the broadcast of process zero. This program may deadlock. The broadcast call on process zero *may* block until process one executes the matching broadcast call, so that the send is not executed. Process one will definitely block on the receive and so, in this case, never executes the broadcast.

The relative order of execution of collective operations and point-to-point operations should be such, so that even if the collective operations and the point-to-point operations are synchronizing, no deadlock will occur.

Example 6.28 An unsafe, non-deterministic program.

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        break;
    case 2:
        MPI_Send(buf2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

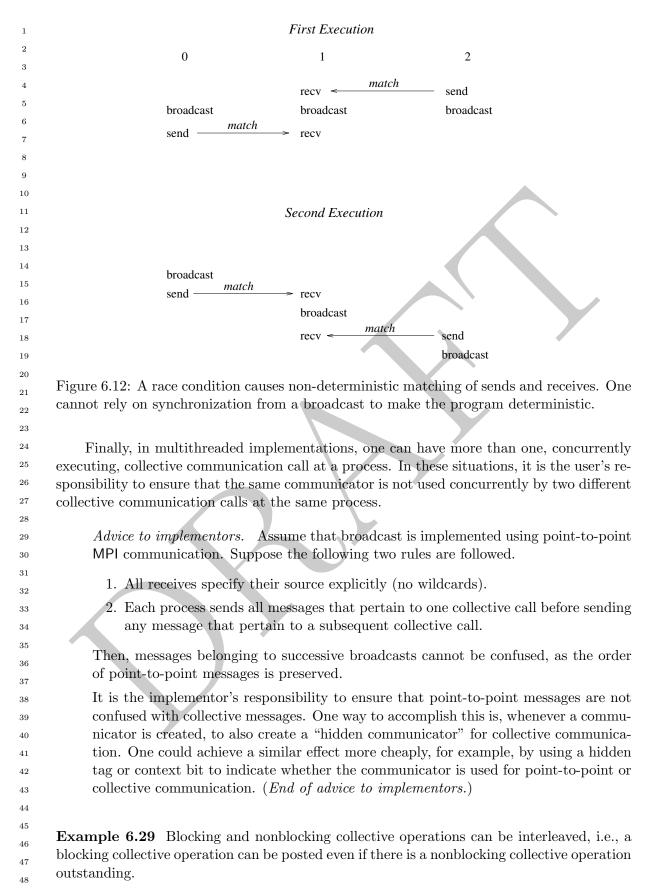
All three processes participate in a broadcast. Process 0 sends a message to process 1 after the broadcast, and process 2 sends a message to process 1 before the broadcast. Process 1 receives before and after the broadcast, with a wildcard source argument.

Two possible executions of this program, with different matchings of sends and receives, are illustrated in Figure 6.12. Note that the second execution has the peculiar effect that a send executed after the broadcast is received at another node before the broadcast. This example illustrates the fact that one should not rely on collective communication functions to have particular synchronization effects. A program that works correctly only when the first execution occurs (only when broadcast is synchronizing) is erroneous.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24



```
MPI_Request req;
```

MPI_Ibarrier(comm, &req); MPI_Bcast(buf1, count, type, 0, comm); MPI_Wait(&req, MPI_STATUS_IGNORE);

Each process starts a nonblocking barrier operation, participates in a blocking broadcast and then waits until every other process started the barrier operation. This effectively turns the broadcast into a synchronizing broadcast with possible communication/communication overlap (MPI_Bcast is allowed, but not required to synchronize).

Example 6.30 The starting order of collective operations on a particular communicator defines their matching. The following example shows an erroneous matching of different collective operations on the same communicator.

```
MPI_Request req;
switch(rank) {
    case 0:
        /* erroneous matching */
        MPI_Ibarrier(comm, &req);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break:
    case 1:
        /* erroneous matching */
        MPI_Bcast(buf1, count, type,
                                     0, comm);
        MPI_Ibarrier(comm, &req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
}
```

This ordering would match MPI_Ibarrier on rank 0 with MPI_Bcast on rank 1 which is erroneous and the program behavior is undefined. However, if such an order is required, the user must create different duplicate communicators and perform the operations on them. If started with two processes, the following program would be correct:

```
36
MPI_Request req;
                                                                                      37
MPI_Comm dupcomm;
                                                                                      38
MPI_Comm_dup(comm, &dupcomm);
                                                                                      39
switch(rank) {
                                                                                      40
    case 0:
                                                                                      41
        MPI_Ibarrier(comm, &req);
                                                                                      42
        MPI_Bcast(buf1, count, type, 0, dupcomm);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
                                                                                      43
                                                                                      44
        break;
                                                                                      45
    case 1:
                                                                                      46
        MPI_Bcast(buf1, count, type, 0, dupcomm);
                                                                                      47
        MPI_Ibarrier(comm, &req);
                                                                                      48
        MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Unofficial Draft for Comment Only

1

2 3

4

5 6

7

8

9

10 11

12

13

14 15

16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30 31

32

33

34

}

break;

Advice to users. The use of different communicators offers some flexibility regarding the matching of nonblocking collective operations. In this sense, communicators could be used as an equivalent to tags. However, communicator construction might induce overheads so that this should be used carefully. (End of advice to users.)

Example 6.31 Nonblocking collective operations can rely on the same progression rules as nonblocking point-to-point messages. Thus, if started with two processes, the following program is a valid MPI program and is guaranteed to terminate:

```
MPI_Request req;
```

```
switch(rank) {
    case 0:
        MPI_Ibarrier(comm, &req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        MPI_Send(buf, count, dtype, 1, tag, comm);
        break;
    case 1:
        MPI_Ibarrier(comm, &req);
        MPI_Recv(buf, count, dtype, 0, tag, comm, MPI_STATUS_IGNORE);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
}
```

The MPI library must progress the barrier in the MPI_Recv call. Thus, the MPI_Wait call in rank 0 will eventually complete, which enables the matching MPI_Send so all calls eventually return.

Example 6.32 Blocking and nonblocking collective operations do not match. The following example is erroneous.

```
<sup>35</sup> MPI_Request req;
```

```
37
     switch(rank) {
38
         case 0:
39
           /* erroneous false matching of Alltoall and Ialltoall */
40
           MPI_Ialltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req);
41
           MPI_Wait(&req, MPI_STATUS_IGNORE);
42
           break;
43
         case 1:
44
           /* erroneous false matching of Alltoall and Ialltoall */
45
           MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);
46
           break;
47
     }
48
```

1

 $\mathbf{2}$

3

 $\frac{4}{5}$

6

7

9

10

11

12

13

14

15

16

17 18

19

20

21 22

23 24

25

26

27

28

29

30

31 32

33

34

35

36

37

38

39

40 41

Example 6.33 Collective and point-to-point requests can be mixed in functions that enable multiple completions. If started with two processes, the following program is valid.

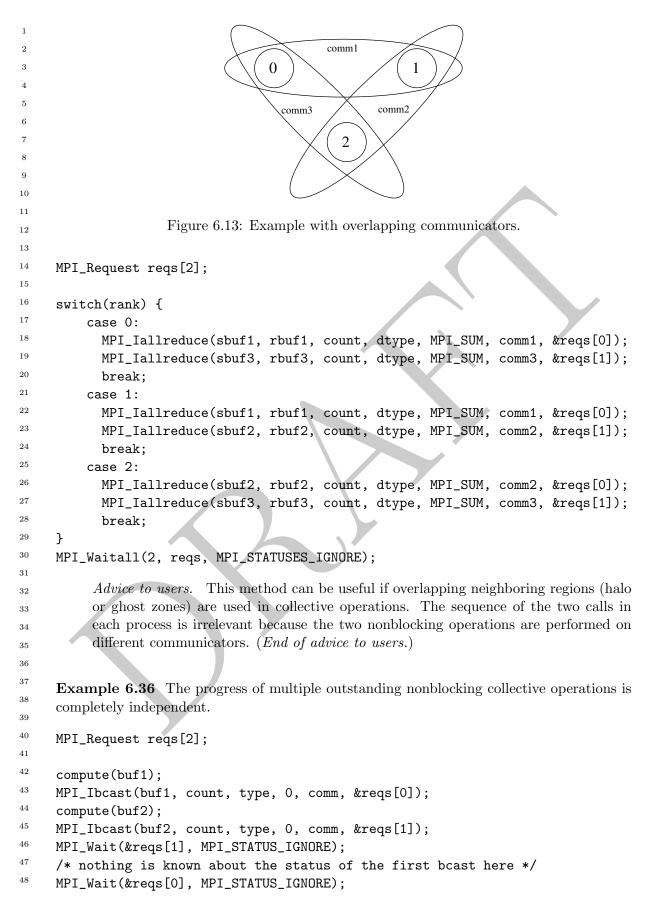
```
MPI_Request reqs[2];
switch(rank) {
    case 0:
      MPI_Ibarrier(comm, &reqs[0]);
      MPI_Send(buf, count, dtype, 1, tag, comm);
      MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
      break:
    case 1:
      MPI_Irecv(buf, count, dtype, 0, tag, comm, &reqs[0]);
      MPI_Ibarrier(comm, &reqs[1]);
      MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
      break;
}
    The MPI_Waitall call returns only after the barrier and the receive completed.
Example 6.34 Multiple nonblocking collective operations can be outstanding on a single
communicator and match in order.
MPI_Request reqs[3];
compute(buf1);
MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
compute(buf2);
MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
```

compute(buf3); MPI_Ibcast(buf3, count, type, 0, comm, &reqs[2]); MPI_Waitall(3, reqs, MPI_STATUSES_IGNORE);

Advice to users. Pipelining and double-buffering techniques can efficiently be used to overlap computation and communication. However, having too many outstanding requests might have a negative impact on performance. (End of advice to users.)

Advice to implementors. The use of pipelining may generate many outstanding requests. A high-quality hardware-supported implementation with limited resources should be able to fall back to a software implementation if its resources are exhausted. In this way, the implementation could limit the number of outstanding requests only by the available memory. (End of advice to implementors.)

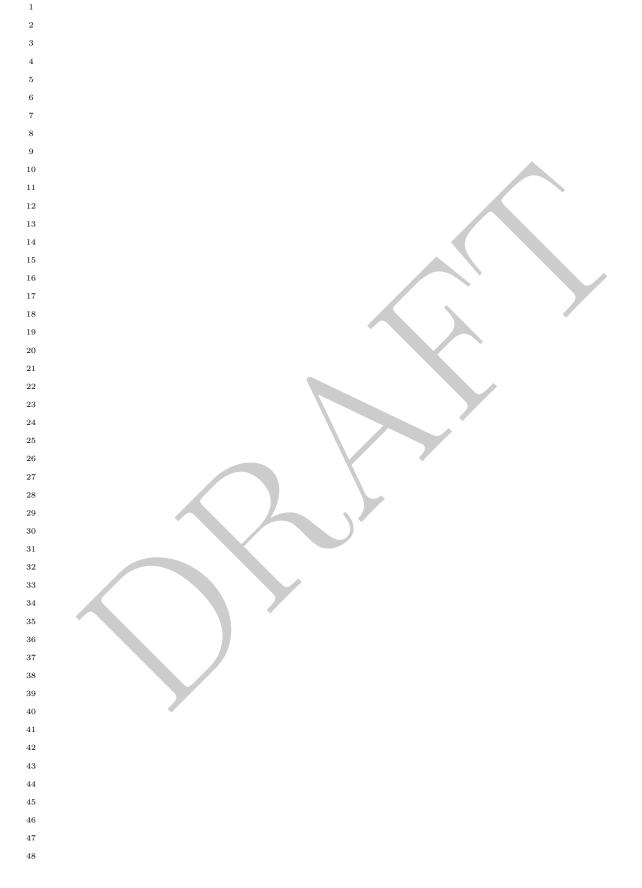
Example 6.35 Nonblocking collective operations can also be used to enable simultaneous collective operations on multiple overlapping communicators (see Figure 6.13). The following example is started with three processes and three communicators. The first communicator comm1 includes ranks 0 and 1, comm2 includes ranks 1 and 2, and comm3 spans ranks 0 and 2. It is not possible to perform a blocking collective operation on all communicators because there exists no deadlock-free order to invoke them. However, nonblocking collective operations can easily be used to achieve this task.



Finishing the second MPI_IBCAST is completely independent of the first one. This means that it is not guaranteed that the first broadcast operation is finished or even started after the second one is completed via reqs[1].

 2

Unofficial Dr	aft for	$\mathbf{Comment}$	Only
---------------	---------	--------------------	------



Chapter 7

Groups, Contexts, Communicators, and Caching

7.1 Introduction

This chapter introduces MPI features that support the development of parallel libraries. Parallel libraries are needed to encapsulate the distracting complications inherent in parallel implementations of key algorithms. They help to ensure consistent correctness of such procedures, and provide a "higher level" of portability than MPI itself can provide. As such, libraries prevent each programmer from repeating the work of defining consistent data structures, data layouts, and methods that implement key algorithms (such as matrix operations). Since the best libraries come with several variations on parallel systems (different data layouts, different strategies depending on the size of the system or problem, or type of floating point), this too needs to be hidden from the user.

We refer the reader to [62] and [4] for further information on writing libraries in MPI, using the features described in this chapter.

7.1.1 Features Needed to Support Libraries

The key features needed to support the creation of robust parallel libraries are as follows:

- Safe communication space, that guarantees that libraries can communicate as they need to, without conflicting with communication extraneous to the library,
- Group scope for collective operations, that allow libraries to avoid unnecessarily synchronizing uninvolved processes (potentially running unrelated code),
- Abstract process naming to allow libraries to describe their communication in terms suitable to their own data structures and algorithms,
- The ability to "adorn" a set of communicating processes with additional user-defined attributes, such as extra collective operations. This mechanism should provide a means for the user or library writer effectively to extend a message-passing notation.

In addition, a unified mechanism or object is needed for conveniently denoting communication context, the group of communicating processes, to house abstract process naming, and to store adornments. 24

7.1.2 MPI's Support for Libraries

The corresponding concepts that MPI provides, specifically to support robust libraries, are as follows:

- **Contexts** of communication,
- Groups of processes,
- Virtual topologies,
- Attribute caching,
- Communicators.

Communicators (see [22, 60, 64]) encapsulate all of these ideas in order to provide the appropriate scope for all communication operations in MPI. Communicators are divided into two kinds: intra-communicators for operations within a single group of processes and inter-communicators for operations between two groups of processes.

¹⁹ Caching. Communicators (see below) provide a "caching" mechanism that allows one to ²⁰ associate new attributes with communicators, on par with MPI built-in features. This can ²¹ be used by advanced users to adorn communicators further, and by MPI to implement ²² some communicator functions. For example, the virtual-topology functions described in ²³ Chapter 8 are likely to be supported this way.

24

1

2

3

4 5

6

7 8

9 10

11

12 13

14

15

16

17 18

Groups. Groups define an ordered collection of processes, each with a rank, and it is this group that defines the low-level names for inter-process communication (ranks are used for sending and receiving). Thus, groups define a scope for process names in point-to-point communication. In addition, groups define the scope of collective operations. Groups may be manipulated separately from communicators in MPI, but only communicators can be used in communication operations.

31

36

37

38

39

40

41

42

43

44

45 46

Intra-communicators. The most commonly used means for message passing in MPI is via
 intra-communicators. Intra-communicators contain an instance of a group, contexts of
 communication for both point-to-point and collective communication, and the ability to
 include virtual topology and other attributes. These features work as follows:

• **Contexts** provide the ability to have separate safe "universes" of message-passing in MPI. A context is akin to an additional tag that differentiates messages. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code. Pending point-to-point communications are also guaranteed not to interfere with collective communications within a single communicator.

- **Groups** define the participants in the communication (see above) of a communicator.
- 47 48

- A virtual topology defines a special mapping of the ranks in a group to and from a topology. Special constructors for communicators are defined in Chapter 8 to provide this feature. Intra-communicators as described in this chapter do not have topologies.
- Attributes define the local information that the user or library has added to a communicator for later reference.

Advice to users. The practice in many communication libraries is that there is a unique, predefined communication universe that includes all processes available when the parallel program is initiated; the processes are assigned consecutive ranks. Participants in a point-to-point communication are identified by their rank; a collective communication (such as broadcast) always involves all processes. When using the World Model (Section 11.2), this practice can be followed in MPI by using the predefined communicator MPI_COMM_WORLD. Users who are satisfied with this practice can plug in MPI_COMM_WORLD wherever a communicator argument is required, and can consequently disregard the rest of this chapter. (End of advice to users.)

Inter-communicators. The discussion has dealt so far with intra-communication: communication within a group. MPI also supports inter-communication: communication between two non-overlapping groups. When an application is built by composing several parallel modules, it is convenient to allow one module to communicate with another using local ranks for addressing within the second module. This is especially convenient in a client-server computing paradigm, where either client or server are parallel. The support of inter-communication also provides a mechanism for the extension of MPI to a dynamic model where not all processes are preallocated at initialization time. In such a situation, it becomes necessary to support communication across "universes." Inter-communication is supported by objects called **inter-communicators**. These objects bind two groups together with communication contexts shared by both groups. For inter-communicators, these features work as follows:

- Contexts provide the ability to have a separate safe "universe" of message-passing between the two groups. A send in the local group is always a receive in the remote group, and vice versa. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code.
- A local and remote group specify the recipients and destinations for an inter-communicator.
- Virtual topology is undefined for an inter-communicator.
- As before, attributes cache defines the local information that the user or library has added to a communicator for later reference.

MPI provides mechanisms for creating and manipulating inter-communicators. They ⁴⁶ are used for point-to-point and collective communication in an related manner to intracommunicators. Users who do not need inter-communication in their applications can safely ⁴⁸

Unofficial Draft for Comment Only

 24

 31

ignore this extension. Users who require inter-communication between overlapping groups must layer this capability on top of MPI.

7.2 Basic Concepts

In this section, we turn to a more formal definition of the concepts introduced above.

7.2.1 Groups

1

 $\mathbf{2}$

3 4

5 6

> 7 8

9

20

21

22

23

24 25

26

27

28

34

35

¹⁰ A **group** is an ordered set of process identifiers (henceforth processes); processes are imple-¹¹ mentation-dependent objects. Each process in a group is associated with an integer **rank**. ¹² Ranks are contiguous and start from zero. Groups are represented by opaque **group ob-**¹³ **jects**, and hence cannot be directly transferred from one process to another. A group is ¹⁴ used within a communicator to describe the participants in a communication "universe" ¹⁵ and to rank such participants (thus giving them unique names within that "universe" of ¹⁶ communication).

¹⁷ There is a special pre-defined group: MPI_GROUP_EMPTY, which is a group with no ¹⁸ members. The predefined constant MPI_GROUP_NULL is the value used for invalid group ¹⁹ handles.

Advice to users. MPI_GROUP_EMPTY, which is a valid handle to an empty group, should not be confused with MPI_GROUP_NULL, which in turn is an invalid handle. The former may be used as an argument to group operations; the latter, which is returned when a group is freed, is not a valid argument. (*End of advice to users.*)

- Advice to implementors. A group may be represented by a virtual-to-real processaddress-translation table. Each communicator object (see below) would have a pointer to such a table.
- Simple implementations of MPI will enumerate groups, such as in a table. However,
 more advanced data structures make sense in order to improve scalability and memory
 usage with large numbers of processes. Such implementations are possible with MPI.
 (*End of advice to implementors.*)
 - 7.2.2 Contexts

A context is a property of communicators (defined next) that allows partitioning of the communication space. A message sent in one context cannot be received in another context. Furthermore, where permitted, collective operations are independent of pending point-topoint operations. Contexts are not explicit MPI objects; they appear only as part of the realization of communicators (below).

 $41 \\ 42$

43

44

45

46

Advice to implementors. Distinct communicators in the same process have distinct contexts. A context is essentially a system-managed tag (or tags) needed to make a communicator safe for point-to-point and MPI-defined collective communication. Safety means that collective and point-to-point communication within one communicator do not interfere, and that communication over distinct communicators don't interfere.

A possible implementation for a context is as a supplemental tag attached to messages on send and matched on receive. Each intra-communicator stores the value of its two tags (one for point-to-point and one for collective communication). Communicatorgenerating functions use a collective communication to agree on a new group-wide unique context.

Analogously, in inter-communication, two context tags are stored per communicator, one used by group A to send and group B to receive, and a second used by group B to send and for group A to receive.

Since contexts are not explicit objects, other implementations are also possible. (*End of advice to implementors.*)

7.2.3 Intra-Communicators

Intra-communicators bring together the concepts of group and context. To support implementation-specific optimizations, and application topologies (defined in the next chapter, Chapter 8), communicators may also "cache" additional information (see Section 7.7). MPI communication operations reference communicators to determine the scope and the "communication universe" in which a point-to-point or collective operation is to operate.

Each communicator contains a group of valid participants; this group always includes the local process. The source and destination of a message is identified by process rank within that group.

For collective communication, the intra-communicator specifies the set of processes that participate in the collective operation (and their order, when significant). Thus, the communicator restricts the "spatial" scope of communication, and provides machine-independent process addressing through ranks.

Intra-communicators are represented by opaque **intra-communicator objects**, and hence cannot be directly transferred from one process to another.

7.2.4 Predefined Intra-Communicators

When using the World Model for MPI initialization, an initial intra-communicator MPI_COMM_WORLD of all processes the local process can communicate with after initialization (itself included) is defined once MPI_INIT or MPI_INIT_THREAD has been called. In addition, the communicator MPI_COMM_SELF is provided, which includes only the process itself. When using the Sessions Model (Section 11.3) for initialization of MPI resources, MPI_COMM_WORLD and MPI_COMM_SELF are not valid for use as a communicator. See the discussion concerning use of MPI named constants in 2.5.4 for valid uses of MPI_COMM_WORLD and MPI_COMM_SELF prior to initialization of MPI.

The predefined constant MPI_COMM_NULL is the value used for invalid communicator handles.

In a static-process-model implementation of MPI, all processes that participate in the computation are available after MPI is initialized. For this case, MPI_COMM_WORLD is a communicator of all processes available for the computation; this communicator has the same value in all processes. In an implementation of MPI where processes can dynamically join an MPI execution, it may be the case that a process starts an MPI computation without having access to all other processes. In such situations, MPI_COMM_WORLD is a communicator incorporating all processes with which the joining process can immediately

279

1

 $\mathbf{2}$

3

4

5

6

7

8

9

10

11 12

13 14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28 29

30

31

32 33

34

35

36 37

38 39

40

48

```
1
     communicate. Therefore, MPI_COMM_WORLD may simultaneously represent disjoint groups
\mathbf{2}
      in different processes.
3
          All MPI implementations are required to provide the MPI_COMM_WORLD communi-
4
      cator. It cannot be deallocated during the life of a process. The group corresponding to
\mathbf{5}
      this communicator does not appear as a pre-defined constant, but it may be accessed using
6
      MPI_COMM_GROUP (see below). MPI does not specify the correspondence between the
\overline{7}
      process rank in MPI_COMM_WORLD and its (machine-dependent) absolute address. Neither
8
      does MPI specify the function of the host process, if any. Other implementation-dependent,
9
     predefined communicators may also be provided.
10
11
      7.3
            Group Management
12
13
      This section describes the manipulation of process groups in MPI. These operations are
14
      local and their execution does not require interprocess communication.
15
16
     7.3.1 Group Accessors
17
18
19
      MPI_GROUP_SIZE(group, size)
20
21
       IN
                                              group (handle)
                 group
22
       OUT
                                               number of processes in the group (integer)
                 size
23
^{24}
      C binding
25
     int MPI_Group_size(MPI_Group group, int *size)
26
27
     Fortran 2008 binding
28
     MPI_Group_size(group, size, ierror)
29
          TYPE(MPI_Group), INTENT(IN) :: group
30
          INTEGER, INTENT(OUT) :: size
^{31}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
      Fortran binding
33
     MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
34
          INTEGER GROUP, SIZE, IERROR
35
36
37
      MPI_GROUP_RANK(group, rank)
38
39
       IN
                                              group (handle)
                  group
40
       OUT
                                               rank of the calling process in group, or
                 rank
41
                                               MPI_UNDEFINED if the process is not a member
42
                                               (integer)
43
44
      C binding
45
      int MPI_Group_rank(MPI_Group group, int *rank)
46
47
      Fortran 2008 binding
48
```

<pre>MPI_Group_rank(group, rank, ierror) TYPE(MPI_Group), INTENT(IN) :: group INTEGER, INTENT(OUT) :: rank INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_GROUP_RANK(GROUP, RANK, IERROR) INTEGER GROUP, RANK, IERROR</pre>			1 2 3 4 5 6 7 8 9
			10
		up1, n, ranks1, group2, ranks2)	11
IN	group1	group1 (handle)	12 13
IN	n	number of ranks in <code>ranks1</code> and <code>ranks2</code> arrays (integer)	14
IN	ranks1	array of zero or more valid ranks in group1	15
IN	group2	group2 (handle)	16
OUT	ranks2	array of corresponding ranks in group2,	17
		MPI_UNDEFINED when no correspondence exists.	18 19
			20
C binding	-		21
int MPI_C	-	_Group group1, int n, const int ranks1[],	22
	MPI_Group group2, in	t ranks2[])	23
	2008 binding		24
-		n, ranks1, group2, ranks2, ierror)	25 26
TYPE(MPI_Group), INTENT(IN) :: group1, group2 INTEGER, INTENT(IN) :: n, ranks1(n)			
INTEGER, INTENT(OUT) :: ranks2(n)			
	GER, OPTIONAL, INTENT(OUT		29
Fortran k	ainding		30
	3	N, RANKS1, GROUP2, RANKS2, IERROR)	31 32
INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR			
This f	unction is important for dotor	nining the relative numbering of the same processes	33 34
This function is important for determining the relative numbering of the same processes in two different groups. For instance, if one knows the ranks of certain processes in the group			
		to know their ranks in a subset of that group.	36
		nput to MPI_GROUP_TRANSLATE_RANKS, which	37
returns MF	PI_PROC_NULL as the translat	ed rank.	38 39
			40
MPI_GRO	UP_COMPARE(group1, group2	2, result)	41
IN	group1	first group (handle)	42
IN	group2	second group (handle)	43
OUT	result		44
001	ICSUIL	result (integer)	45 46
C binding			
int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result) 44			
	1 - 1 - 1 P - 0		

1	Fortran 2	008 binding			
2	MPI_Group_compare(group1, group2, result, ierror)				
3	TYPE(MPI_Group), INTENT(IN) :: group1, group2				
4	INTEGER, INTENT(OUT) :: result				
5			INTENT(OUT) :: ierror		
6	INIEG	ER, OFFENRE,			
7	Fortran b	oinding			
8	MPI_GROUP	COMPARE(GROUP	1, GROUP2, RESULT, IERROR)		
9	INTEG	ER GROUP1, GRO	UP2, RESULT, IERROR		
		- 1			
10			up members and group order is exactly the same in both groups.		
11			$group1$ and $group2$ are the same handle. $MPI_SIMILAR$ results if		
12	the group 1	members are the s	ame but the order is different. MPI_UNEQUAL results otherwise.		
13					
14	7.3.2 Gro	oup Constructors			
15					
16	-		s to constructing groups. In the first approach, MPI procedures		
17	-		superset existing groups. These constructors construct new		
18	· ·		s. In the second approach, a group is created using a session		
19		-	ss set. This second approach is available when using the Sessions		
20		* *	ches, these are local operations, and distinct groups may be		
21	defined on	different processe	s; a process may also define a group that does not include itself.		
22	Consistent	definitions are re	equired when groups are used as arguments in communicator-		
23	building fu	nctions. When us	sing the World Model for initializing MPI, the base group, upon		
24	which all o	other groups are o	defined, is the group associated with the initial communicator		
25	MPI_COMM	M_WORLD (accessi	ible through the function MPI_COMM_GROUP).		
26					
27	Ratio	onale. In what	follows, there is no group duplication function analogous to		
28	MPI_	_COMM_DUP, de	fined later in this chapter. There is no need for a group dupli-		
29	cator	A group, once	created, can have several references to it by making copies of		
30	the h	andle. The follow	ving constructors address the need for subsets and supersets of		
31	exist	ing groups. (End	of rationale.)		
32					
33	Advi	ce to implemento	rs. Each group constructor behaves as if it returned a new		
34	grou	p object. When	this new group is a copy of an existing group, then one can		
35	avoid	l creating such ne	ew objects, using a reference-count mechanism. (End of advice		
36	to in	nplementors.)			
37					
38			`		
39	MPI_COM	M_GROUP(comm	, group)		
40 41	IN	comm	communicator (handle)		
42	OUT	group	group corresponding to comm (handle)		
43					
44	C binding	r 5			
45	int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)				
46					
47	Fortran 2008 binding				
48	MPI_Comm_	group(comm, gr	oup, ierror)		

TYPE	E(MPI_Comm), INTENT()	IN) :: comm	1
TYPE(MPI_Group), INTENT(OUT) :: group			2
INTE	EGER, OPTIONAL, INTE	NT(OUT) :: ierror	3
Fortran	binding		4
Fortran binding MPI_COMM_GROUP(COMM, GROUP, IERROR)		5	
	EGER COMM, GROUP, IE		6
			7
MPI.	_COMM_GROUP return	is in group a handle to the group of comm.	8 9
			10
MPI_GR0	OUP_UNION(group1, gro	pup2, newgroup)	11
IN	group1	first group (handle)	12
			13
IN	group2	second group (handle)	14
OUT	newgroup	union group (handle)	15
			16
C bindi	0		17
int MPI_	-	up group1, MPI_Group group2,	18 19
	MPI_Group *new	group)	19 20
Fortran	2008 binding		20
		up2, newgroup, ierror)	22
		(IN) :: group1, group2	23
	E(MPI_Group), INTENT		24
TN.L.F	EGER, OPTIONAL, INTE	NT(OUT) :: ierror	25
Fortran	binding		26
MPI_GROU	JP_UNION(GROUP1, GROU	UP2, NEWGROUP, IERROR)	27
INTE	EGER GROUP1, GROUP2,	NEWGROUP, IERROR	28
			29
			30
MPI_GR0	OUP_INTERSECTION(g	roup1, group2, newgroup)	31 32
IN	group1	first group (handle)	33
IN	group2	second group (handle)	34
			35
OUT	newgroup	intersection group (handle)	36
			37
C bindi	0	MPI_Group group1, MPI_Group group2,	38
IIIC MFI_	MPI_Group *new		39
	· -	Bronh	40
	2008 binding		41
		p1, group2, newgroup, ierror)	42 43
	-	(IN) :: group1, group2	43 44
	E(MPI_Group), INTENT EGER, OPTIONAL, INTE		44
		NI(001) TETTOT	46
Fortran	-		47
MPI_GROU	JP_INTERSECTION(GROU	P1, GROUP2, NEWGROUP, IERROR)	48

```
1
          INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
\mathbf{2}
3
4
      MPI_GROUP_DIFFERENCE(group1, group2, newgroup)
5
       IN
                 group1
                                               first group (handle)
6
       IN
                  group2
                                               second group (handle)
7
8
       OUT
                 newgroup
                                               difference group (handle)
9
10
     C binding
11
      int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
12
                     MPI_Group *newgroup)
13
     Fortran 2008 binding
14
     MPI_Group_difference(group1, group2, newgroup, ierror)
15
          TYPE(MPI_Group), INTENT(IN) :: group1, group2
16
          TYPE(MPI_Group), INTENT(OUT) :: newgroup
17
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
      Fortran binding
20
     MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
21
          INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
22
23
      The set-like operations are defined as follows:
^{24}
      union All elements of the first group (group1), followed by all elements of second group
25
           (group2) not in the first group.
26
      intersect all elements of the first group that are also in the second group, ordered as in
27
           the first group.
28
29
      difference all elements of the first group that are not in the second group, ordered as in
30
           the first group.
^{31}
32
      Note that for these operations the order of processes in the output group is determined
      primarily by order in the first group (if possible) and then, if necessary, by order in the
33
34
      second group. Neither union nor intersection are commutative, but both are associative.
          The new group can be empty, that is, equal to MPI_GROUP_EMPTY.
35
36
37
      MPI_GROUP_INCL(group, n, ranks, newgroup)
38
39
       IN
                                               group (handle)
                  group
40
       IN
                                               number of elements in array ranks (and size of
                  n
41
                                               newgroup) (integer)
42
       IN
                                               ranks of processes in group to appear in newgroup
                  ranks
43
                                               (array of integers)
44
       OUT
                                               new group derived from above, in the order defined
                  newgroup
45
                                               by ranks (handle)
46
47
48
      C binding
```

Fortran 2008 binding

```
MPI_Group_incl(group, n, ranks, newgroup, ierror)
   TYPE(MPI_Group), INTENT(IN) :: group
   INTEGER, INTENT(IN) :: n, ranks(n)
   TYPE(MPI_Group), INTENT(OUT) :: newgroup
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

The function MPI_GROUP_INCL creates a group newgroup that consists of the n processes in group with ranks ranks[0],..., ranks[n-1]; the process with rank i in newgroup is the process with rank ranks[i] in group. Each of the n elements of ranks must be a valid rank in group and all elements must be distinct, or else the program is erroneous. If n = 0, then newgroup is MPI_GROUP_EMPTY. This function can, for instance, be used to reorder the elements of a group. See also MPI_GROUP_COMPARE.

MPI_GROUP_EXCL(group, n, ranks, newgroup)

			22
IN	group	group (handle)	23
IN	n	number of elements in array ranks (integer)	24
IN	ranks	array of integer ranks of processes in group not to	25
		appear in newgroup	26
			27
OUT	newgroup	new group derived from above, preserving the order	28
		defined by group (handle)	29
			30
C binding	g		31
int MPI_G	roup_excl(MPI_Group grou	p, int n, const int ranks[],	32
	MPI_Group *newgroup))	33
Fortron 2	008 binding		34
	e e	······	35
-	_excl(group, n, ranks, n	• ·	36
	<pre>MPI_Group), INTENT(IN) :</pre>		37
	ER, INTENT(IN) :: n, ran		38
	MPI_Group), INTENT(OUT)		39
INTEG	ER, OPTIONAL, INTENT(OUT) :: ierror	40
Fortran b	inding		41
	_EXCL(GROUP, N, RANKS, N	EWGROUP, IERROR)	42
INTEG	ER GROUP, N, RANKS(*), N	EWGROUP, IERROR	43
TTI C			44
The fu		reates a group of processes newgroup that is obtained	45

The function MPI_GROUP_EXCL creates a group of processes newgroup that is obtained by deleting from group those processes with ranks ranks[0],..., ranks[n-1]. The ordering of processes in newgroup is identical to the ordering in group. Each of the n elements of ranks

286 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
1
      must be a valid rank in group and all elements must be distinct; otherwise, the program is
\mathbf{2}
      erroneous. If n = 0, then newgroup is identical to group.
3
4
      MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)
5
6
        IN
                                                  group (handle)
                   group
7
        IN
                                                  number of triplets in array ranges (integer)
                   n
8
        IN
                   ranges
                                                  a one-dimensional array of integer triplets, of the
9
                                                  form (first rank, last rank, stride) indicating ranks in
10
                                                  group of processes to be included in newgroup
11
                                                  new group derived from above, in the order defined
12
        OUT
                   newgroup
13
                                                  by ranges (handle)
14
15
      C binding
16
      int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
17
                       MPI_Group *newgroup)
18
      Fortran 2008 binding
19
      MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
20
           TYPE(MPI_Group), INTENT(IN) :: group
21
           INTEGER, INTENT(IN) :: n, ranges(3, n)
22
           TYPE(MPI_Group), INTENT(OUT) :: newgroup
23
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
25
      Fortran binding
26
      MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
27
           INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR
28
      If ranges consists of the triplets

(first_1, last_1, stride_1), \dots, (first_n, last_n, stride_n)
29
30
^{31}
32
      then newgroup consists of the sequence of processes in group with ranks
33
           first_1, first_1 + stride_1, \dots, first_1 + \left| \frac{last_1 - first_1}{stride_1} \right| stride_1, \dots,
34
35
           first_n, first_n + stride_n, \dots, first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.
36
37
38
39
           Each computed rank must be a valid rank in group and all computed ranks must be
40
      distinct, or else the program is erroneous. Note that we may have first_i > last_i, and stride_i
41
      may be negative, but cannot be zero.
42
           The functionality of this routine is specified to be equivalent to expanding the array
43
      of ranges to an array of the included ranks and passing the resulting array of ranks and
44
      other arguments to MPI_GROUP_INCL. A call to MPI_GROUP_INCL is equivalent to a call
45
      to MPI_GROUP_RANGE_INCL with each rank i in ranks replaced by the triplet (i,i,1) in the
46
      argument ranges.
47
```

MPI_GRUU	JP_RANGE_EXCL(group, n, ra	inges, newgroup)	-
IN	group	group (handle)	2
IN	n	number of triplets in array ranges (integer)	4
IN	ranges	a one-dimensional array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group of processes to be excluded from the output group newgroup (array of integers)	5 7 8
OUT	newgroup	new group derived from above, preserving the order in group (handle)	9 1 1

MPL GROUP RANGE EXCL(group n ranges newgroup)

C binding

```
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
             MPI_Group *newgroup)
```

Fortran 2008 binding

MPI_Group_range_excl(group, n, ranges, newgroup, ierror)
TYPE(MPI_Group), INTENT(IN) :: group
<pre>INTEGER, INTENT(IN) :: n, ranges(3, n)</pre>
TYPE(MPI_Group), INTENT(OUT) :: newgroup
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_	_GROUP_R	ANGE_	EXCL((GROUP,	N,	RANGE	ΞS,	NEWGRO	JUP,	IERROR)
	INTEGER	GROU	P, N,	RANGES	3(3,	*),	NEW	IGROUP	, IEF	ROR	

Each computed rank must be a valid rank in group and all computed ranks must be distinct, or else the program is erroneous.

The functionality of this routine is specified to be equivalent to expanding the array of ranges to an array of the excluded ranks and passing the resulting array of ranks and other arguments to MPI_GROUP_EXCL. A call to MPI_GROUP_EXCL is equivalent to a call to MPI_GROUP_RANGE_EXCL with each rank i in ranks replaced by the triplet (i,i,1) in the argument ranges.

The range operations do not explicitly enumerate ranks, and Advice to users. therefore are more scalable if implemented efficiently. Hence, we recommend MPI programmers to use them whenenever possible, as high-quality implementations will take advantage of this fact. (End of advice to users.)

Advice to implementors. The range operations should be implemented, if possible, without enumerating the group members, in order to obtain better scalability (time and space). (End of advice to implementors.)

 31

```
288 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING
```

```
1
     MPI_GROUP_FROM_SESSION_PSET(session, pset_name, newgroup)
2
       IN
                session
                                            session (handle)
3
       IN
                                            name of process set to use to create the new group
                 pset_name
4
                                            (string)
5
6
       OUT
                                            new group derived from supplied session and process
                newgroup
7
                                            set (handle)
8
9
     C binding
10
     int MPI_Group_from_session_pset(MPI_Session session, const char *pset_name,
11
                    MPI_Group *newgroup)
12
     Fortran 2008 binding
13
     MPI_Group_from_session_pset(session, pset_name, newgroup, ierror)
14
         TYPE(MPI_Session), INTENT(IN) :: session
15
         CHARACTER(LEN=*), INTENT(IN) :: pset_name
16
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     Fortran binding
20
     MPI_GROUP_FROM_SESSION_PSET(SESSION, PSET_NAME, NEWGROUP, IERROR)
21
          INTEGER SESSION, NEWGROUP, IERROR
22
         CHARACTER*(*) PSET_NAME
23
         The function MPI_GROUP_FROM_SESSION_PSET creates a group newgroup using the
^{24}
     provided session handle and process set. The process set name must be one returned from
25
     an invocation of MPI_SESSION_GET_NTH_PSET using the supplied session handle. If the
26
     pset_name does not exist, MPI_GROUP_NULL will be returned in the newgroup argument.
27
     As with other group constructors, MPI_GROUP_FROM_SESSION_PSET is a local function.
28
     See Section 11.3 for more information on sessions and process sets.
29
30
^{31}
     7.3.3 Group Destructors
32
33
34
     MPI_GROUP_FREE(group)
35
       INOUT group
                                            group (handle)
36
37
     C binding
38
     int MPI_Group_free(MPI_Group *group)
39
40
     Fortran 2008 binding
41
     MPI_Group_free(group, ierror)
42
         TYPE(MPI_Group), INTENT(INOUT) :: group
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     Fortran binding
45
     MPI_GROUP_FREE(GROUP, IERROR)
46
         INTEGER GROUP, IERROR
47
48
```

This operation marks a group object for deallocation. The handle group is set to MPI_GROUP_NULL by the call. Any on-going operation using this group will complete normally.

Advice to implementors. One can keep a reference count that is incremented for each call to MPI_COMM_GROUP, MPI_COMM_CREATE, MPI_COMM_DUP, and MPI_COMM_IDUP, and decremented for each call to MPI_GROUP_FREE or MPI_COMM_FREE; the group object is ultimately deallocated when the reference count drops to zero. (*End of advice to implementors.*)

7.4 Communicator Management

This section describes the manipulation of communicators in MPI. Operations that access communicators are local and their execution does not require interprocess communication. Operations that create communicators are collective and may require interprocess communication.

Advice to implementors. High-quality implementations should amortize the overheads associated with the creation of communicators (for the same group, or subsets thereof) over several calls, by allocating multiple contexts with one collective communication. (End of advice to implementors.)

7.4.1 Communicator Accessors		23
		24
The following are all local operations.		25
		26
MPI_COMM_SIZE(comm, size)		27
		28
IN comm	communicator (handle)	29
OUT size	number of processes in the group of comm (integer)	30
		31
C binding		32
int MPI_Comm_size(MPI_Comm comm, in	nt xgiza)	33
THE MILCOMM_SIZE(MILCOMM COMM, II		34
Fortran 2008 binding		35
<pre>MPI_Comm_size(comm, size, ierror)</pre>		36
TYPE(MPI_Comm), INTENT(IN) :: (comm	37
INTEGER, INTENT(OUT) :: size		38
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	39
Fouture his diam		40
Fortran binding		41
MPI_COMM_SIZE(COMM, SIZE, IERROR)		42
INTEGER COMM, SIZE, IERROR		43
		44
Rationale. This function is equiva	lent to accessing the communicator's group with	45

Rationale. This function is equivalent to accessing the communicator's group with ⁴⁵ MPI_COMM_GROUP (see above), computing the size using MPI_GROUP_SIZE, and ⁴⁶ then freeing the temporary group via MPI_GROUP_FREE. However, this function is ⁴⁷ so commonly used that this shortcut was introduced. (*End of rationale.*) ⁴⁸

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

 $\overline{7}$

8

9 10 11

12 13

14

15

16

17 18

19

20

21

```
1
           Advice to users.
                               This function indicates the number of processes involved in a
2
           communicator. For MPI_COMM_WORLD, it indicates the total number of processes
3
           available unless the number of processes has been changed by using the functions
4
           described in Chapter 11; note that the number of processes in MPI_COMM_WORLD
5
           does not change during the life of an MPI program.
6
           This call is often used with the next call to determine the amount of concurrency
7
           available for a specific library or program. The following call, MPI_COMM_RANK
8
           indicates the rank of the process that calls it in the range from 0, \ldots, size-1, where
9
           size is the return value of MPI_COMM_SIZE.(End of advice to users.)
10
11
12
     MPI_COMM_RANK(comm, rank)
13
14
       IN
                                              communicator (handle)
                 comm
15
       OUT
                 rank
                                              rank of the calling process in group of comm (integer)
16
17
     C binding
18
     int MPI_Comm_rank(MPI_Comm comm, int *rank)
19
20
     Fortran 2008 binding
21
     MPI_Comm_rank(comm, rank, ierror)
22
          TYPE(MPI_Comm), INTENT(IN) :: comm
23
          INTEGER, INTENT(OUT) :: rank
24
          INTEGER, OPTIONAL, INTENT(OUT)
                                              :: ierror
25
26
     Fortran binding
     MPI_COMM_RANK(COMM, RANK, IERROR)
27
          INTEGER COMM, RANK, IERROR
28
29
30
           Rationale.
                       This function is equivalent to accessing the communicator's group with
31
           MPI_COMM_GROUP (see above), computing the rank using MPI_GROUP_RANK,
32
           and then freeing the temporary group via MPI_GROUP_FREE. However, this function
33
           is so commonly used that this shortcut was introduced. (End of rationale.)
34
35
           Advice to users. This function gives the rank of the process in the particular commu-
36
           nicator's group. It is useful, as noted above, in conjunction with MPI_COMM_SIZE.
37
           Many programs will be written with the supervisor/executor or manager/worker
38
           model, where one process (such as the rank-zero process) will play a supervisory
39
           role, and the other processes will serve as compute nodes. In this framework, the
40
           two preceding calls are useful for determining the roles of the various processes of a
41
           communicator. (End of advice to users.)
42
43
44
45
46
47
48
```

MPL COMM_COMPARE(comm1, comm2, result)

IN	comm1	first communicator (handle)	
IN	comm2	second communicator (handle)	
OUT	result	result (integer)	

C binding

int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)

Fortran 2008 binding

```
MPI_Comm_compare(comm1, comm2, result, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
    INTEGER, INTENT(OUT) :: result
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR) INTEGER COMM1, COMM2, RESULT, IERROR

MPI_IDENT results if and only if comm1 and comm2 are handles for the same object (identical groups and same contexts). MPI_CONGRUENT results if the underlying groups are identical in constituents and rank order; these communicators differ only by context. MPI_SIMILAR results if the group members of both communicators are the same but the rank order differs. MPI_UNEQUAL results otherwise.

7.4.2 Communicator Constructors

The following are collective functions that are invoked by all processes in the group or groups associated with comm, with the exception of MPI_COMM_CREATE_GROUP, MPI_COMM_CREATE_FROM_GROUP, and MPI_INTERCOMM_CREATE_FROM_GROUPS. MPI_COMM_CREATE_GROUP and MPI_COMM_CREATE_FROM_GROUP are invoked only by the processes in the group of the new communicator being constructed. MPI_INTERCOMM_CREATE_FROM_GROUPS is invoked by all the processes in the local and remote groups of the new communicator being constructed. See the discussion below for the definition of local and remote groups.

Rationale. Note that, when using the World Model, there is a chicken-and-egg aspect to MPI in that a communicator is needed to create a new communicator. In the World Model, the base communicator for all MPI communicators is predefined outside of MPI, and is MPI_COMM_WORLD. The World Model was arrived at after considerable debate, and was chosen to increase "safety" of programs written in MPI. (*End of rationale.*)

This chapter presents the following communicator construction routines: MPI_COMM_CREATE, MPI_COMM_DUP, MPI_COMM_IDUP, MPI_COMM_DUP_WITH_INFO, MPI_COMM_IDUP_WITH_INFO and MPI_COMM_SPLIT can be used to create both intracommunicators and intercommunicators; MPI_COMM_CREATE_GROUP, MPI_COMM_CREATE_FROM_GROUP, and MPI_INTERCOMM_MERGE (see Section 7.6.2) can be used to create intracommunicators;

```
1
     MPI_INTERCOMM_CREATE and MPI_INTERCOMM_CREATE_FROM_GROUPS (see Sec-
\mathbf{2}
     tion 7.6.2) can be used to create intercommunicators.
3
          An intracommunicator involves a single group while an intercommunicator involves
4
     two groups. Where the following discussions address intercommunicator semantics, the
\mathbf{5}
     two groups in an intercommunicator are called the left and right groups. A process in an
6
     intercommunicator is a member of either the left or the right group. From the point of view
7
     of that process, the group that the process is a member of is called the local group; the
8
     other group (relative to that process) is the remote group. The left and right group labels
9
     give us a way to describe the two groups in an intercommunicator that is not relative to
10
     any particular process (as the local and remote groups are).
11
12
     MPI_COMM_DUP(comm, newcomm)
13
14
       IN
                                              communicator (handle)
                 comm
15
       OUT
                                              copy of comm (handle)
                 newcomm
16
17
     C binding
18
     int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
19
20
     Fortran 2008 binding
21
     MPI_Comm_dup(comm, newcomm, ierror)
22
          TYPE(MPI_Comm), INTENT(IN) :: comm
23
          TYPE(MPI_Comm), INTENT(OUT) :: newcomm
^{24}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     Fortran binding
26
     MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
27
          INTEGER COMM, NEWCOMM, IERROR
28
29
          MPI_COMM_DUP duplicates the existing communicator comm with associated key
30
     values and topology information. For each key value, the respective copy callback function
^{31}
     determines the attribute value associated with this key in the new communicator; one
32
     particular action that a copy callback may take is to delete the attribute from the new
33
     communicator. MPI_COMM_DUP returns in newcomm a new communicator with the same
34
     group or groups, same topology, and any copied cached information, but a new context (see
35
     Section 7.7.1).
36
37
           Advice to users. This operation is used to provide a parallel library with a duplicate
38
           communication space that has the same properties as the original communicator. This
           includes any attributes (see below) and topologies (see Chapter 8). This call is valid
39
40
           even if there are pending point-to-point communications involving the communicator
41
           comm. A typical call might involve a MPI_COMM_DUP at the beginning of the
42
           parallel call, and an MPI_COMM_FREE of that duplicated communicator at the end
           of the call. Other models of communicator management are also possible.
43
44
           This call applies to both intra- and inter-communicators. (End of advice to users.)
45
46
           Advice to implementors. One need not actually copy the group information, but only
47
           add a new reference and increment the reference count. Copy on write can be used
48
           for the cached information. (End of advice to implementors.)
```

MPI_CON	MM_DUP_WITH_IN	FO(comm, info, newcomm)	1
IN	comm	communicator (handle)	2 3
IN	info	info object (handle)	4
OUT	newcomm	copy of comm (handle)	5
C bindin int MPI_	•	fo(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)	6 7 8 9
MPI_Comm TYPE TYPE INTE Fortran MPI_COMM	C(MPI_Comm), INTE C(MPI_Info), INTE C(MPI_Comm), INTE CGER, OPTIONAL, I binding	NT(IN) :: info NT(OUT) :: newcomm NTENT(OUT) :: ierror OMM, INFO, NEWCOMM, IERROR)	9 10 11 12 13 14 15 16 17 18 19
hints prov Rat time an i	vided by the argument ionale. It is expect e. However, for legation nfo argument. One	H_INFO behaves exactly as MPI_COMM_DUP except that the nt info are associated with the output communicator newcomm. ed that some hints will only be valid at communicator creation acy reasons, most communicator creation calls do not provide may associate info hints with a duplicate of any communicator h a call to MPI_COMM_DUP_WITH_INFO. (<i>End of rationale.</i>)	 20 21 22 23 24 25 26 27 28
MPI_CON	/IM_IDUP(comm, ne	ewcomm, request)	29 30
IN	comm	communicator (handle)	31
OUT	newcomm	copy of comm (handle)	32
OUT	request	communication request (handle)	33
			34 35
C bindir	ıg		36
int MPI_	Comm_idup(MPI_Co	mm comm, MPI_Comm *newcomm, MPI_Request *request)	37
Fortran	2008 binding		38
		omm, request, ierror)	39
TYPE	C(MPI_Comm), INTE	NT(IN) :: comm	40
		NT(OUT), ASYNCHRONOUS :: newcomm	41
	-	NTENT(OUT) :: request	42 43
TN.LE	GER, UPTIUNAL, I	NTENT(OUT) :: ierror	43 44
Fortran	binding		45
	-	OMM, REQUEST, IERROR)	46
INTE	GER COMM, NEWCOM	M, REQUEST, IERROR	47
			48

1 2 3 4 5	of its nont was execut after MPL	locking behavior, ted at the time tha _COMM_IDUP wit	nonblocking variant of MPI_COMM_DUP. With the exception the semantics of MPI_COMM_IDUP are as if MPI_COMM_DUP at MPI_COMM_IDUP is called. For example, attributes changed Il not be copied to the new communicator. All restrictions and
6	-	1M_IDUP and the	g collective operations (see Section 6.12) apply to
7			he communicator newcomm as an input argument to other MPI
8			OMM_IDUP operation completes.
9			
10 11	MPI_COM	IM_IDUP_WITH_	INFO(comm, info, newcomm, request)
12	IN	comm	communicator (handle)
13	IN	info	info object (handle)
14	OUT	newcomm	copy of comm (handle)
15 16	OUT		communication request (handle)
17	001	request	communication request (nandle)
18	C bindin	Q	
19			info(MPI_Comm comm, MPI_Info info,
20 21		MPI_Comm *	newcomm, MPI_Request *request)
21	Fortran 2	2008 binding	
23			(comm, info, newcomm, request, ierror)
24	TYPE	(MPI_Comm), INT	ENT(IN) :: comm
25	TYPE	(MPI_Info), INT	ENT(IN) :: info
26			ENT(OUT), ASYNCHRONOUS :: newcomm
27		-	INTENT(OUT) :: request
28	INTE	GER, OPTIONAL,	INTENT(OUT) :: ierror
29	Fortran l	binding	
30	MPI_COMM	_IDUP_WITH_INFO	(COMM, INFO, NEWCOMM, REQUEST, IERROR)
31 32	INTE	GER COMM, INFO,	NEWCOMM, REQUEST, IERROR
33	MPI_	COMM_IDUP_WI	TH_INFO is a nonblocking variant of
34			NFO. With the exception of its nonblocking behavior, the se-
35	mantics of	f MPI_COMM_ID	UP_WITH_INFO are as if MPI_COMM_DUP_WITH_INFO was
36	executed a	t the time that MI	PI_COMM_IDUP_WITH_INFO is called. For example, attributes
37		9	MPI_COMM_IDUP_WITH_INFO will not be copied to the new
38			ons and assumptions for nonblocking collective operations (see
39		/ = = / -	COMM_IDUP_WITH_INFO and the returned request.
40			he communicator newcomm as an input argument to other MPI
41	functions	before the MPI_C	OMM_IDUP_WITH_INFO operation completes.
42 43	Rati	onale. The MPI	_COMM_IDUP and MPI_COMM_IDUP_WITH_INFO functions
43			velopment of purely nonblocking libraries (see [40]). (End of
45		onale.)	
46		,	
47			
48			

	(, 8+, 8+,	
IN	comm	communicator (handle)
IN	group	group, which is a subset of the group of comm (handle)
OUT	newcomm	new communicator (handle)

MPI_COMM_CREATE(comm, group, newcomm)

C binding

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

Fortran 2008 binding

MPI_Comm_create(comm, group, newcomm, ierror)
 TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Group), INTENT(IN) :: group
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR) INTEGER COMM, GROUP, NEWCOMM, IERROR

If comm is an intracommunicator, this function returns a new communicator newcomm with communication group defined by the group argument. No cached information propagates from comm to newcomm. Each process must call MPI_COMM_CREATE with a group argument that is a subgroup of the group associated with comm; this could be MPI_GROUP_EMPTY. The processes may specify different values for the group argument. If a process calls with a non-empty group then all processes in that group must call the function with the same group as argument, that is the same processes in the same order. Otherwise, the call is erroneous. This implies that the set of groups specified across the processes must be disjoint. If the calling process is a member of the group given as group argument, then newcomm is a communicator with group as its associated group. In the case that a process calls with a group to which it does not belong, e.g., MPI_GROUP_EMPTY, then MPI_COMM_NULL is returned as newcomm. The function is collective and must be called by all processes in the group of comm.

Rationale. The interface supports the original mechanism from MPI-1.1, which required the same group in all processes of comm. It was extended in MPI-2.2 to allow the use of disjoint subgroups in order to allow implementations to eliminate unnecessary communication that MPI_COMM_SPLIT would incur when the user already knows the membership of the disjoint subgroups. (*End of rationale.*)

Rationale. The requirement that the entire group of comm participate in the call stems from the following considerations:

- It allows the implementation to layer MPI_COMM_CREATE on top of regular collective communications.
- It provides additional safety, in particular in the case where partially overlapping groups are used to create new communicators.
- It permits implementations to sometimes avoid communication related to context creation.

Unofficial Draft for Comment Only

2

5 6

9

10

11

12

13

14

15

16 17

18

19

20

21

22

23

 24

25

26

27

28

29

30

 31

32

33 34

35

36

37

38

39

40

41

42

43

44

45

46 47

296 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

1	(End of rationale.)					
2						
3	Advice to users. MPI_COMM_CREATE provides a means to subset a group of pro-					
4	cesses for the purpose of separate MIMD computation, with separate communication					
5	space. newcomm, which emerges from MPI_COMM_CREATE, can be used in subse-					
6	quent calls to MPI_COMM_CREATE (or other communicator constructors) to further					
7	subdivide a computation into parallel sub-computations. A more general service is					
8	provided by MPI_COMM_SPLIT, below. (End of advice to users.)					
9 10	Advice to implementors. When calling MPI_COMM_DUP, all processes call with the					
11	same group (the group associated with the communicator). When calling					
12	MPI_COMM_CREATE, the processes provide the same group or disjoint subgroups.					
13	For both calls, it is theoretically possible to agree on a group-wide unique context					
14	with no communication. However, local execution of these functions requires use					
15	of a larger context name space and reduces error checking. Implementations may					
16	strike various compromises between these conflicting goals, such as bulk allocation of					
17	multiple contexts in one collective operation.					
18	Important: If new communicators are created without synchronizing the processes					
19	involved then the communication system must be able to cope with messages arriving					
20	in a context that has not yet been allocated at the receiving process. (End of advice					
21	to implementors.)					
22						
23	If comm is an intercommunicator, then the output communicator is also an intercommun-					
24	icator where the local group consists only of those processes contained in group (see Fig-					
25	ure 7.1). The group argument should only contain those processes in the local group of					
26	the input intercommunicator that are to be a part of newcomm . All processes in the same					
27 28	local group of comm must specify the same value for group, i.e., the same members in the same order. If either group does not specify at least one process in the local group of the					
29	intercommunicator, or if the calling process is not included in the group, MPI_COMM_NULL					
30	is returned.					
31						
32	Rationale. In the case where either the left or right group is empty, a null communi-					
33	cator is returned instead of an intercommunicator with MPI_GROUP_EMPTY because					
34	the side with the empty group must return MPI_COMM_NULL. (End of rationale.)					
35						
36	Example 7.1 Inter-communicator creation.					
37	The following example illustrates how the first node in the left side of an intercommunicator					
38	could be joined with all members on the right side of an intercommunicator to form a new					
39	intercommunicator.					
40						
41	<pre>MPI_Comm inter_comm, new_inter_comm;</pre>					
42	MPI_Group local_group, group;					
43	int rank = 0; /* rank on left side to include in					
44 45	new inter-comm */					
45	/* Construct the original intercommunications winter communications					
47	<pre>/* Construct the original intercommunicator: "inter_comm" */</pre>					
48						

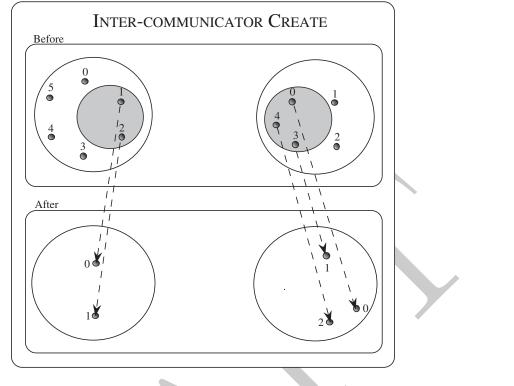


Figure 7.1: Intercommunicator creation using MPI_COMM_CREATE extended to intercommunicators. The input groups are those in the grey circle.

	/* Construct the group of	processes to be in new	25	
	<pre>/* Construct the group of processes to be in new intercommunicator */</pre>			
if (/* I'm on the left side of the intercommunicator */) {				
<pre>MPI_Comm_group(inter_comm, &local_group); MPI_Group_incl(local_group, 1, &rank, &group);</pre>				
			30	
	MPI_Group_free(&local_g	roup);	31	
			32	
	else		33	
	MPI_Comm_group(inter_com	nm, &group);	34	
			35	
		n, group, &new_inter_comm);	36	
<pre>MPI_Group_free(&group);</pre>				
			39	
MPI_COM	IM_CREATE_GROUP(comm, g	group, tag, newcomm)	40	
IN	comm	intracommunicator (handle)	41	
IN	group	group, which is a subset of the group of comm	42	
	8	(handle)	43	
			44	
IN	tag	tag (integer)	45	
OUT	newcomm	new communicator (handle)	46	
			47	
C bindin	C binding			

C binding

 $\mathbf{2}$

 23

 24

1 2	int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag, MPI_Comm *newcomm)
3 4 5 6 7 8 9 10	<pre>Fortran 2008 binding MPI_Comm_create_group(comm, group, tag, newcomm, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Group), INTENT(IN) :: group INTEGER, INTENT(IN) :: tag TYPE(MPI_Comm), INTENT(OUT) :: newcomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
11 12 13	Fortran binding MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR) INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR
14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30	MPI_COMM_CREATE_GROUP is similar to MPI_COMM_CREATE; however, MPI_COMM_CREATE must be called by all processes in the group of comm, whereas MPI_COMM_CREATE_GROUP must be called by all processes in group, which is a subgroup of the group of comm. In addition, MPI_COMM_CREATE_GROUP requires that comm is an intracommunicator. MPI_COMM_CREATE_GROUP returns a new intracommunicator, newcomm, for which the group argument defines the communication group. No cached infor- mation propagates from comm to newcomm. Each process must provide a group argument that is a subgroup of the group associated with comm; this could be MPI_GROUP_EMPTY. If a non-empty group is specified, then all processes in that group must call the function, and each of these processes must provide the same arguments, including a group that contains the same members with the same ordering. Otherwise the call is erroneous. If the calling process is a member of the group given as the group argument, then newcomm is a commu- nicator with group as its associated group. If the calling process is not a member of group, e.g., group is MPI_GROUP_EMPTY, then the call is a local operation and MPI_COMM_NULL is returned as newcomm.
30 31 32 33 34 35 36 37	Rationale. Functionality similar to MPI_COMM_CREATE_GROUP can be imple- mented through repeated MPI_INTERCOMM_CREATE and MPI_INTERCOMM_MERGE calls that start with the MPI_COMM_SELF communicators at each process in group and build up an intracommunicator with group group [17]. Such an algorithm requires the creation of many intermediate communicators; MPI_COMM_CREATE_GROUP can provide a more efficient implementation that avoids this overhead. (<i>End of rationale.</i>)
38 39 40 41 42	Advice to users. An intercommunicator can be created collectively over processes in the union of the local and remote groups by creating the local communicator using MPI_COMM_CREATE_GROUP and using that communicator as the local communicator argument to MPI_INTERCOMM_CREATE. (<i>End of advice to users.</i>)
43 44 45 46 47 48	The tag argument does not conflict with tags used in point-to-point communication and is not permitted to be a wildcard. If multiple threads at a given process perform concurrent MPI_COMM_CREATE_GROUP operations, the user must distinguish these operations by providing different tag or comm arguments.

Advice to users. MPI_COMM_CREATE may provide lower overhead than MPI_COMM_CREATE_GROUP because it can take advantage of collective communication on comm when constructing newcomm. (*End of advice to users.*)

MPI_COMM_SPLIT(comm, color, key, newcomm)

IN	comm	communicator (handle)
IN	color	control of subset assignment (integer)
IN	key	control of rank assignment (integer)
OUT	newcomm	new communicator (handle)

C binding

int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

Fortran 2008 binding

MPI_Comm_split(comm, color, key, newcomm, ierror)
 TYPE(MPI_Comm), INTENT(IN) :: comm
 INTEGER, INTENT(IN) :: color, key
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_	_COMM_SF	PLIT(CO	MM, COI	LOR,	KEY,	NEWCO	MM,	IERR	OR)
	INTEGER	COMM,	COLOR	, KEY	, NEV	VCOWM.	IEF	ROR	

This function partitions the group associated with comm into disjoint subgroups, one for each value of color. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. A process may supply the color value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL. This is a collective call, but each process is permitted to provide different values for color and key.

With an intracommunicator comm, a call to MPI_COMM_CREATE(comm, group, newcomm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where processes that are members of their group argument provide color = number of the group (based on a unique numbering of all disjoint groups) and key = rank in group, and all processes that are not members of their group argument provide color = MPI_UNDEFINED.

The value of color must be non-negative or MPI_UNDEFINED.

This is an extremely powerful mechanism for dividing a single Advice to users. 41 communicating group of processes into k subgroups, with k chosen implicitly by the 42user (by the number of colors asserted over all the processes). Each resulting com-43 municator will be non-overlapping. Such a division could be useful for defining a 44 hierarchy of computations, such as for multigrid, or linear algebra. For intracommu-45nicators, MPI_COMM_SPLIT provides similar capability as MPI_COMM_CREATE to 46 split a communicating group into disjoint subgroups. MPI_COMM_SPLIT is useful 47when some processes do not have complete information of the other members in their 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4 5 6

15

16

17

18

19

20

21

22 23

24 25 26

27

28

29

30

 31

32

33

34

35

36

37

38

¹ group, but all processes know (the color of) the group to which they belong. In this ² case, the MPI implementation discovers the other group members via communication. ³ MPI_COMM_CREATE is useful when all processes have complete information of the ⁴ members of their group. In this case, MPI can avoid the extra communication required ⁵ to discover group membership. MPI_COMM_CREATE_GROUP is useful when all pro-⁶ cesses in a given group have complete information of the members of their group and ⁷ synchronization with processes outside the group can be avoided.

- ⁸ Multiple calls to MPI_COMM_SPLIT can be used to overcome the requirement that ⁹ any call have no overlap of the resulting communicators (each process is of only one ¹⁰ color per call). In this way, multiple overlapping communication structures can be ¹¹ created. Creative use of the color and key in such splitting operations is encouraged.
- Note that, for a fixed color, the keys need not be unique. It is MPI_COMM_SPLIT's responsibility to sort processes in ascending order according to this key, and to break ties in a consistent way. If all the keys are specified in the same way, then all the processes in a given color will have the relative rank order as they did in their parent group.
- Essentially, making the key value zero for all processes of a given color means that one
 does not really care about the rank-order of the processes in the new communicator.
 (*End of advice to users.*)
 - *Rationale.* color is restricted to be non-negative, so as not to confict with the value assigned to MPI_UNDEFINED. (*End of rationale.*)

The result of MPI_COMM_SPLIT on an intercommunicator is that those processes on the left with the same color as those processes on the right combine to create a new intercommunicator. The key argument describes the relative rank of processes on each side of the intercommunicator (see Figure 7.2). For those colors that are specified only on one side of the intercommunicator, MPI_COMM_NULL is returned. MPI_COMM_NULL is also returned to those processes that specify MPI_UNDEFINED as the color.

Advice to users. For intercommunicators, MPI_COMM_SPLIT is more general than MPI_COMM_CREATE. A single call to MPI_COMM_SPLIT can create a set of disjoint intercommunicators, while a call to MPI_COMM_CREATE creates only one. (*End of advice to users.*)

³⁶₃₇ **Example 7.2** Parallel client-server model.

21

22

23

31

32

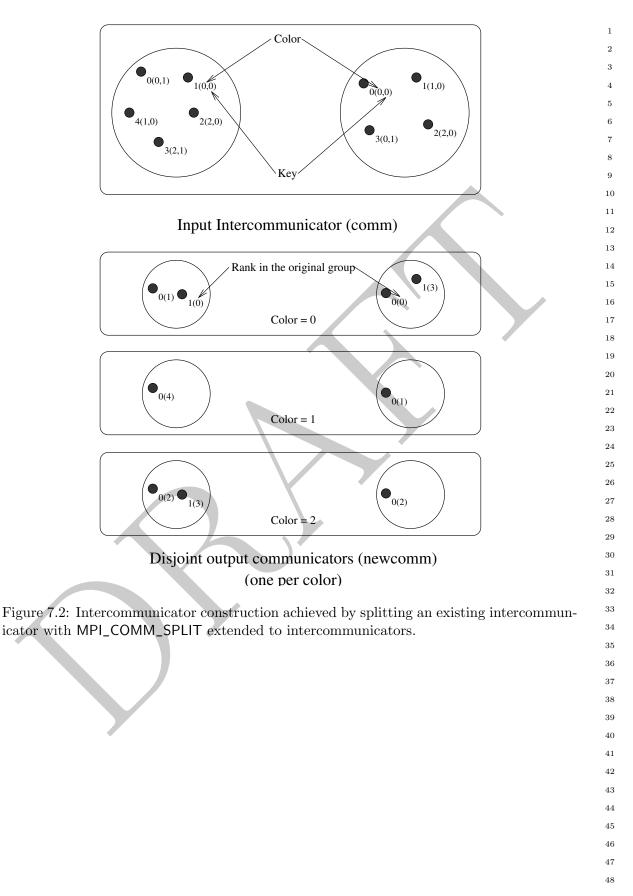
33

34

35

The following client code illustrates how clients on the left side of an intercommunicator could be assigned to a single server from a pool of servers on the right side of an intercommunicator.

```
41
              /* Client code */
42
             MPI_Comm multiple_server_comm;
43
             MPI_Comm
                        single_server_comm;
44
              int
                         color, rank, num_servers;
45
46
              /* Create intercommunicator with clients and servers:
47
                 multiple_server_comm */
48
              . . .
```



```
1
2
              /* Find out the number of servers available */
3
              MPI_Comm_remote_size(multiple_server_comm, &num_servers);
4
5
              /* Determine my color */
6
              MPI_Comm_rank(multiple_server_comm, &rank);
7
              color = rank % num_servers;
8
9
              /* Split the intercommunicator */
10
              MPI_Comm_split(multiple_server_comm, color, rank,
11
                              &single_server_comm);
12
     The following is the corresponding server code:
13
14
              /* Server code */
15
              MPI_Comm multiple_client_comm;
16
              MPI_Comm single_server_comm;
17
              int
                         rank;
18
19
              /* Create intercommunicator with clients and servers:
20
                 multiple_client_comm */
21
              . . .
22
23
              /* Split the intercommunicator for a single server per group
24
                 of clients */
25
              MPI_Comm_rank(multiple_client_comm, &rank);
26
              MPI_Comm_split(multiple_client_comm, rank, 0,
27
                              &single_server_comm);
28
29
30
     MPI_COMM_SPLIT_TYPE(comm, split_type, key, info, newcomm)
^{31}
32
       IN
                                            communicator (handle)
                comm
33
       IN
                split_type
                                            type of processes to be grouped together (integer)
34
       1N
                key
                                            control of rank assignment (integer)
35
36
       INOUT
                info
                                           info argument (handle)
37
       OUT
                newcomm
                                            new communicator (handle)
38
39
     C binding
40
     int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,
41
                   MPI_Info info, MPI_Comm *newcomm)
42
43
     Fortran 2008 binding
44
     MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
45
         TYPE(MPI_Comm), INTENT(IN) :: comm
46
         INTEGER, INTENT(IN) :: split_type, key
47
         TYPE(MPI_Info), INTENT(IN) :: info
48
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)
INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR

This function partitions the group associated with comm into disjoint subgroups such that each subgroup contains all MPI processes in the same grouping referred to by split_type. Within each subgroup, the MPI processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. This is a collective call. All MPI processes in the group associated with comm must provide the same split_type, but each MPI process is permitted to provide different values for key. An exception to this rule is that an MPI process may supply the type value MPI_UNDEFINED, in which case MPI_COMM_NULL is returned in newcomm for such MPI process.

For split_type, the following values are defined by MPI:

MPI_COMM_TYPE_SHARED — all MPI processes in newcomm can create a shared memory segment (e.g., with a successful call to MPI_WIN_ALLOCATE_SHARED). This segment can subsequently be used for load/store accesses by all MPI processes in newcomm.

Advice to users. Since the location of some of the MPI processes may change during the application execution, the communicators created with the value MPI_COMM_TYPE_SHARED before this change may not reflect an actual ability to share memory between MPI processes after this change. (*End of advice to users.*)

MPI_COMM_TYPE_HW_GUIDED — this value specifies that the communicator comm is split according to a **hardware resource type** (for example a computing core or an L3 cache) specified by the "mpi_hw_resource_type" info key. Each output communicator newcomm corresponds to a single instance of the specified hardware resource type. The MPI processes in the group associated with the output communicator newcomm utilize that specific hardware resource type instance, and no other instance of the same hardware resource type.

If an MPI process does not meet the above criteria, then MPI_COMM_NULL is returned in newcomm for such process.

MPI_COMM_NULL is also returned in newcomm in the following cases:

- No info key is provided.
- The info handle does not include the key "mpi_hw_resource_type".
- The MPI implementation neither recognizes nor supports the info key "mpi_hw_resource_type".
- The MPI implementation does not recognize the value associated with the info key "mpi_hw_resource_type".

The MPI implementation will return in the group of the output communicator newcomm the largest subset of MPI processes that match the splitting criterion.

2

5 6

 $\overline{7}$

8

9

10

11

12

13

14

15

16 17

18

19 20

21

22

23

 24

25 26

27

28

29

30

 31

32

33 34

35

36

37

38 39

40

41

42

43

44 45

46

304 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

The processes in the group associated with **newcomm** are ranked in the order defined by the value of the argument **key** with ties broken according to their rank in the group associated with **comm**.

Advice to users. The set of hardware resources that an MPI process is able to utilize may change during the application execution (e.g., because of the relocation of an MPI process), in which case the communicators created with the value MPI_COMM_TYPE_HW_GUIDED before this change may not reflect the utilization of hardware resources of such process at any time after the communicator creation. (*End of advice to users.*)

The user explicitly constrains with the info argument the splitting of the input communicator comm. To this end, the info key "mpi_hw_resource_type" is reserved and its associated value is an implementation-defined string designating the type of the requested hardware resource (e.g., "NUMANode", "Package" or "L3Cache").

The value "mpi_shared_memory" is reserved and its use is equivalent to using MPI_COMM_TYPE_SHARED for the split_type parameter.

Rationale. The value "mpi_shared_memory" is defined in order to ensure consistency between the use of MPI_COMM_TYPE_SHARED and the use of MPI_COMM_TYPE_HW_GUIDED. (*End of rationale.*)

All MPI processes must provide the same value for the info key "mpi_hw_resource_type".

Example 7.3 Splitting MPI_COMM_WORLD into NUMANode subcommunicators.

MPI_Info info; MPI_Comm hwcomm; int rank;

MPI_COMM_TYPE_HW_UNGUIDED — the group of MPI processes associated with newcomm must be a *strict* subset of the group associated with comm and each newcomm corresponds to a single instance of a hardware resource type (for example a computing core or an L3 cache).
 All MPI processes in the group associated with comm which utilize that group for each processes.

All MPI processes in the group associated with **comm** which utilize that specific hardware resource type instance – and no other instance of the same hardware resource type – are included in the group of **newcomm**.

⁴⁶ If a given MPI process cannot be a member of a communicator that forms such a ⁴⁷ strict subset, or does not meet the above criteria, then MPI_COMM_NULL is returned ⁴⁸ in newcomm for this process.

Advice to implementation. In a high-quality MPI implementation, the number of different new valid communicators **newcomm** produced by this splitting operation should be minimal unless the user provides a key/value pair that modifies this behavior. The sets of hardware resource types used for the splitting operation are implementation-dependent, but should reflect the hardware of the actual system on which the application is currently executing. (End of advice to implementors.)

If the hardware resources are hierarchically organized, calling this Rationale. routine several times using as its input communicator comm the output communicator newcomm of the previous call creates a sequence of newcomm communicators in each MPI process, which exposes a hierarchical view of the hardware platform, as shown in Example 7.4. This sequence of returned newcomm communicators may differ from the sets of hardware resource types, as shown in the second splitting operation in Figure 7.3. (End of rationale.)

Advice to users. Each output communicator newcomm can represent a different hardware resource type (see Figure 7.3 for an example). The set of hardware resources an MPI process utilizes may change during the application execution (e.g., because of process relocation), in which case the communicators created with the value MPI_COMM_TYPE_HW_UNGUIDED before this change may not reflect the utilization of hardware resources for such process at any time after the communicator creation. (End of advice to users.)

If a valid info handle is provided as an argument, the MPI implementation sets the info key "mpi_hw_resource_type" for each MPI process in the group associated with a returned newcomm communicator and the info key value is an implementation-defined string that indicates the hardware resource type represented by newcomm. The same hardware resource type must be set in all MPI processes in the group associated with newcomm.

Example 7.4 Recursive splitting of MPI_COMM_WORLD.

			32	
#define MAX_NUM_LEVELS 32				
			34	
MPI_Comm hwcomm [MAX_]		35		
int rank, level_num		36		
			37	
<pre>hwcomm[level_num] = MPI_COMM_WORLD;</pre>				
			39	
while((hwcomm[level_num] != MPI_COMM_NULL)				
&& (level_num < MAX_NUM_LEVELS-1)){				
<pre>MPI_Comm_rank(hwcomm[level_num],&rank);</pre>				
<pre>MPI_Comm_split_type(hwcomm[level_num],</pre>			43	
	MPI_COMM_TYPE_HW_UNGUIDED,		44	
	rank,		45	
	MPI_INFO_NULL,		46	
	<pre>&hwcomm[level_num+1]);</pre>		47	
<pre>level_num++;</pre>			48	

Unofficial Draft for Comment Only

1 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

 $\overline{7}$

8

9

10

11

12

13

1415

16

17

18

19

20

21

22 23

24

25

26

27

28

2930

 31 32

1 } 2 3 4 Advice to implementations. Implementations can define their own split_type values, or 5use the info argument, to assist in creating communicators that help expose platform-6 specific information to the application. The concept of hardware-based communicators 7 was first described by Träff [67] for SMP systems. Guided and unguided modes 8 description as well as an implementation path are introduced by Goglin *et al.* [27]. 9 (End of advice to implementors.) 10 11 1213MPI_COMM_CREATE_FROM_GROUP(group, stringtag, info, errhandler, newcomm) 14IN group group (handle) 15unique identifier for this operation (string) IN 16stringtag 17 IN info info object (handle) 18 IN errhandler error handler to be attached to new 19 intra-communicator (handle) 20OUT new communicator (handle) 21newcomm 2223C binding 24 int MPI_Comm_create_from_group(MPI_Group group, const char *stringtag, 25MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newcomm) 26Fortran 2008 binding 27MPI_Comm_create_from_group(group, stringtag, info, errhandler, newcomm, 28ierror) 29 TYPE(MPI_Group), INTENT(IN) :: group 30 CHARACTER(LEN=*), INTENT(IN) :: stringtag 31TYPE(MPI_Info), INTENT(IN) :: info 32 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler 33 TYPE(MPI_Comm), INTENT(OUT) :: newcomm 34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35 36 Fortran binding 37 MPI_COMM_CREATE_FROM_GROUP(GROUP, STRINGTAG, INFO, ERRHANDLER, NEWCOMM, 38 IERROR) 39 INTEGER GROUP, INFO, ERRHANDLER, NEWCOMM, IERROR 40 CHARACTER*(*) STRINGTAG 41 MPI_COMM_CREATE_FROM_GROUP is similar to MPI_COMM_CREATE_GROUP, ex-42cept that the set of MPI processes involved in the creation of the new intracommunicator 43 is specified by a group argument, rather than the group associated with a pre-existing com-44 municator. If a non-empty group is specified, then all MPI processes in that group must call 45 the function and each of these MPI processes must provide the same arguments, including 46 a group that contains the same members with the same ordering, and identical stringtag 47value. In the event that MPI_GROUP_EMPTY is supplied as the group argument, then the 48

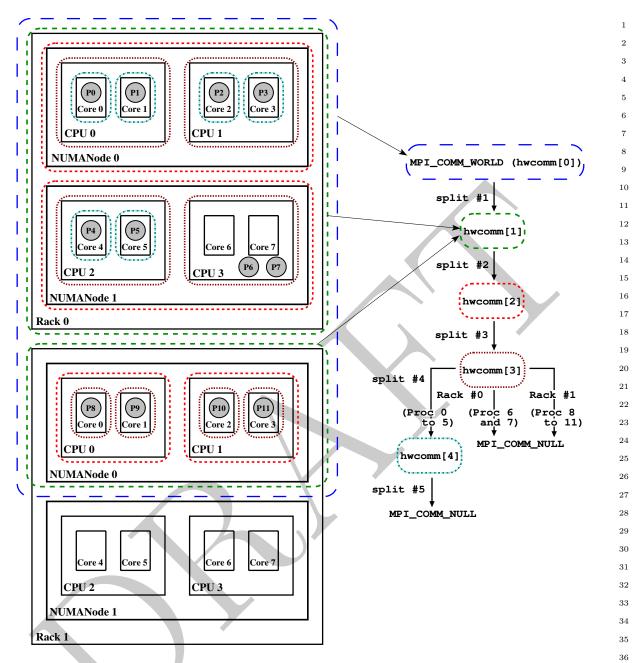


Figure 7.3: Recursive splitting of MPI_COMM_WORLD with MPI_COMM_SPLIT_TYPE and MPI_COMM_TYPE_HW_UNGUIDED. Dashed lines represent communicators whilst solid lines represent hardware resources. MPI processes (P0 to P11) utilize exclusively their respective core, except for P6 and P7 which utilize CPU #3 of Rack #0 and can therefore use Cores #6 and #7 indifferently. The second splitting operation yields two subcommunicators corresponding to NUMANodes in Rack #0 and to CPUs in Rack #1 because Rack #1 features only one NUMANode which corresponds to the whole portion of the Rack that is included in MPI_COMM_WORLD and hwcomm[1]. For the first splitting operation, the hardware resource type returned in the info argument is "Rack" on the processes on Rack #0, whereas on Rack #1, it can be either "Rack" or "NUMANode".

1 call is a local operation and MPI_COMM_NULL is returned as newcomm. The stringtag argu- $\mathbf{2}$ ment is analogous to the tag used for MPI_COMM_CREATE_GROUP. If multiple threads at 3 a given MPI process perform concurrent MPI_COMM_CREATE_FROM_GROUP operations, 4 the user must distinguish these operations by providing different stringtag arguments. The $\mathbf{5}$ stringtag shall not exceed MPI_MAX_FROM_GROUP_TAG characters in length. For C, this 6 includes space for a null terminating character. The errhandler argument specifies an error 7handler to be attached to the new intracommunicator. This error handler will also be in-8 voked if the MPI_COMM_CREATE_FROM_GROUP function encounters an error. The info 9 argument provides hints and assertions, possibly MPI implementation dependent, which 10 indicate desired characteristics and guide communicator creation. 11Advice to users. The stringtag argument is used to distinguish concurrent commu-12nicator construction operations issued by different entities. As such, it is important 13 to ensure that this argument is unique for each concurrent call to 14MPI_COMM_CREATE_FROM_GROUP. Reverse domain name notation convention [1] 15is one approach to constructing unique stringtag arguments. See also example 11.8. 16(End of advice to users.) 17 18 7.4.3 Communicator Destructors 19 2021MPI_COMM_FREE(comm) 22 23INOUT communicator to be destroyed (handle) comm 24 25C binding 26int MPI_Comm_free(MPI_Comm *comm) 27Fortran 2008 binding 28MPI_Comm_free(comm, ierror) 29TYPE(MPI_Comm), INTENT(INOUT) :: comm 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 31 32 Fortran binding 33 MPI_COMM_FREE(COMM, IERROR) 34 INTEGER COMM, IERROR 35 36

This collective operation marks the communication object for deallocation. The handle is set to MPI_COMM_NULL. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it. This call applies to intra- and inter-communicators. The delete callback functions for all cached attributes (see Section 7.7) are called in arbitrary order.

Advice to implementors. Though collective, it is anticipated that this operation will normally be implemented to be local, though a debugging version of an MPI library might choose to synchronize. (*End of advice to implementors.*)

7.4.4 Communicator Info

37

38

39

40 41

42

43

44 45

46

⁴⁷ Hints specified via info (see Chapter 10) allow a user to provide information to direct
 ⁴⁸ optimization. Providing hints may enable an implementation to deliver increased per-

formance or minimize use of system resources. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI_COMM_GET_INFO) and that place a restriction on the behavior of the application. Hints are specified on a per communicator basis, in MPI_COMM_DUP_WITH_INFO, MPI_COMM_IDUP_WITH_INFO, MPI_COMM_SET_INFO, MPI_COMM_SPLIT_TYPE, MPI_DIST_GRAPH_CREATE, and MPI_DIST_GRAPH_CREATE_ADJACENT, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI_COMM_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

Info hints are not propagated by MPI from one communicator to another. The following info keys are valid for all communicators.

- "mpi_assert_no_any_tag" (boolean, default: "false"): If set to "true", then the implementation may assume that the process will not use the MPI_ANY_TAG wildcard on the given communicator.
- "mpi_assert_no_any_source" (boolean, default: "false"): If set to "true", then the implementation may assume that the process will not use the MPI_ANY_SOURCE wildcard on the given communicator.
- "mpi_assert_exact_length" (boolean, default: "false"): If set to "true", then the implementation may assume that the lengths of messages received by the process are equal to the lengths of the corresponding receive buffers, for point-to-point communication operations on the given communicator.
- "mpi_assert_allow_overtaking" (boolean, default: "false"): If set to "true", then the implementation may assume that point-to-point communications on the given communicator do not rely on the non-overtaking rule specified in Section 3.5. In other words, the application asserts that send operations are not required to be matched at the receiver in the order in which the send operations were posted by the sender, and receive operations are not required to be matched in the order in which they were posted by the receiver.

Advice to users. Use of the "mpi_assert_allow_overtaking" info key can result in nondeterminism in the message matching order. (*End of advice to users.*)

Advice to users. Some optimizations may only be possible when all processes in the group of the communicator provide a given info key with the same value. (End of advice to users.)

1 2

3

4

 $\mathbf{5}$

6 7

> 8 9

10

11

12

13

14

15

16 17

18

19 20

21

22

23 24

25

26

27

28

29

30 31

32

33

34

35

36

37

38

39 40

41

42 43

44

45

```
1
     MPI_COMM_SET_INFO(comm, info)
2
       INOUT
                                              communicator (handle)
                 comm
3
       IN
                 info
                                              info object (handle)
4
5
6
     C binding
\overline{7}
     int MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)
8
     Fortran 2008 binding
9
     MPI_Comm_set_info(comm, info, ierror)
10
          TYPE(MPI_Comm), INTENT(IN) :: comm
11
          TYPE(MPI_Info), INTENT(IN) :: info
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_COMM_SET_INFO(COMM, INFO, IERROR)
16
          INTEGER COMM, INFO, IERROR
17
          MPI_COMM_SET_INFO updates the hints of the communicator associated with comm
18
     using the hints provided in info. This operation has no effect on previously set or defaulted
19
     hints that are not specified by info. It also has no effect on previously set or defaulted
20
     hints that are specified by info, but are ignored by the MPI implementation in this call to
21
     MPI_COMM_SET_INFO. MPI_COMM_SET_INFO is a collective routine. The info object
22
     may be different on each process, but any info entries that an implementation requires to
23
     be the same on all processes must appear with the same value in each process's info object.
24
25
                             Some info items that an implementation can use when it creates
           Advice to users.
26
           a communicator cannot easily be changed once the communicator has been created.
27
           Thus, an implementation may ignore hints issued in this call that it would have
28
           accepted in a creation call. An implementation may also be unable to update certain
29
           info hints in a call to MPI_COMM_SET_INFO. MPI_COMM_GET_INFO can be used to
30
           determine whether updates to existing info hints were ignored by the implementation.
31
           (End of advice to users.)
32
33
                              Setting info hints on the predefined communicators
           Advice to users.
34
           MPI_COMM_WORLD and MPI_COMM_SELF may have unintended effects, as changes to
35
           these global objects may affect all components of the application, including libraries
36
           and tools. Users must ensure that all components of the application that use a given
37
           communicator, including libraries and tools, can comply with any info hints associated
38
           with that communicator. (End of advice to users.)
39
40
41
     MPI_COMM_GET_INFO(comm, info_used)
42
43
       IN
                                              communicator object (handle)
                 comm
44
       OUT
                 info_used
                                              new info object (handle)
45
46
     C binding
47
     int MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)
48
```

Fortran 2008 binding

```
MPI_Comm_get_info(comm, info_used, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(OUT) :: info_used
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR) INTEGER COMM, INFO_USED, IERROR

MPI_COMM_GET_INFO returns a new info object containing the hints of the communicator associated with comm. The current setting of all hints related to this communicator is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info_used via MPI_INFO_FREE.

7.5 Motivating Examples 7.5.1 Current Practice #1 Example #1a: int main(int argc, char *argv[]) { int me, size; . . . MPI_Init(&argc, &argv); MPI_Comm_rank(MPI_COMM_WORLD, &me); MPI_Comm_size(MPI_COMM_WORLD, &size); (void)printf("Process %d size %d\n", me, size); . . . MPI_Finalize(); return 0; }

Example #1a is a do-nothing program that initializes itself, and refers to the "all" communicator, and prints a message. It terminates itself too. This example does not imply that MPI supports printf-like communication itself. Example #1b: Message exchange (supposing that size is even)

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

5 6

9 10

11

12 13

14

15

16

17

18 19

20 21

22

23 24

25

26

27

28

29

30

31 32

33

34

35

36

37 38

39

40

41

```
1
\mathbf{2}
             MPI_Comm_rank(MPI_COMM_WORLD, &me);
                                                         /* local */
3
             MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */
4
5
             if((me \% 2) == 0)
6
             {
7
                 /* send unless highest-numbered process */
8
                 if((me + 1) < size)
9
                    MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
10
             }
11
             else
12
                 MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);
13
14
              . . .
15
             MPI_Finalize();
16
             return 0;
17
          }
18
     Example #1b schematically illustrates message exchanges between "even" and "odd" pro-
19
     cesses in the "all" communicator.
20
21
     7.5.2 Current Practice #2
22
23
         int main(int argc, char *argv[])
^{24}
         {
25
           int me, count;
26
           void *data;
27
           . . .
28
29
           MPI_Init(&argc, &argv);
30
           MPI_Comm_rank(MPI_COMM_WORLD, &me);
31
32
           if(me == 0)
33
            Ł
34
                /* get input, create buffer ''data'' */
35
36
           }
37
38
           MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);
39
40
           . . .
^{41}
           MPI_Finalize();
42
           return 0;
         }
43
44
     This example illustrates the use of a collective communication.
45
46
     7.5.3
            (Approximate) Current Practice #3
47
48
        int main(int argc, char *argv[])
```

```
{
  int me, count, count2;
  void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
  MPI_Group group_world, grprem;
  MPI_Comm commWorker;
  static int ranks[] = {0};
  . . .
  MPI_Init(&argc, &argv);
  MPI_Comm_group(MPI_COMM_WORLD, &group_world);
  MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
  MPI_Group_excl(group_world, 1, ranks, &grprem); /* local */
  MPI_Comm_create(MPI_COMM_WORLD, grprem, &commWorker);
  if(me != 0)
  {
    /* compute on worker */
    MPI_Reduce(send_buf,recv_buf,count, MPI_INT, MPI_SUM, 1, commWorker);
    MPI_Comm_free(&commWorker);
  }
  /* zero falls through immediately to this reduce, others do later... */
  MPI_Reduce(send_buf2, recv_buf2, count2,
             MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  MPI_Group_free(&group_world);
  MPI_Group_free(&grprem);
  MPI_Finalize();
  return 0;
}
```

This example illustrates how a group consisting of all but the zeroth process of the "all" group is created, and then how a communicator is formed (commWorker) for that new group. The new communicator is used in a collective call, and all processes execute a collective call in the MPI_COMM_WORLD context. This example illustrates how the two communicators (that inherently possess distinct contexts) protect communication. That is, communication in MPI_COMM_WORLD is insulated from communication in commWorker, and vice versa.

In summary, "group safety" is achieved via communicators because distinct contexts within communicators are enforced to be unique on any process.

7.5.4 Example #4

The following example is meant to illustrate "safety" between point-to-point and collective communication. MPI guarantees that a single communicator can do safe point-to-point and collective communication.

#define TAG_ARBITRARY 12345
#define SOME_COUNT 50

Unofficial Draft for Comment Only

2

6

9

10

11

12

13 14

15 16

17

18

19 20 21

22

23 24

25 26

27

28 29

30 31

32

33

34

35

36

37

38

39

40 41

42 43

44

45

46 47

```
1
\mathbf{2}
        int main(int argc, char *argv[])
3
        ſ
4
           int me;
5
          MPI_Request request[2];
6
           MPI_Status status[2];
7
           MPI_Group group_world, subgroup;
8
           int ranks[] = {2, 4, 6, 8};
9
           MPI_Comm the_comm;
10
           . . .
11
           MPI_Init(&argc, &argv);
12
           MPI_Comm_group(MPI_COMM_WORLD, &group_world);
13
14
          MPI_Group_incl(group_world, 4, ranks, &subgroup); /* local */
15
                                                  /* local */
          MPI_Group_rank(subgroup, &me);
16
17
          MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
18
19
           if(me != MPI_UNDEFINED)
20
           ſ
21
               MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
22
                                   the_comm, request);
23
               MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
^{24}
                                   the_comm, request+1);
25
               for(i = 0; i < SOME_COUNT; i++)</pre>
26
                 MPI_Reduce(..., the_comm);
27
               MPI_Waitall(2, request, status);
28
29
               MPI_Comm_free(&the_comm);
30
           }
31
32
           MPI_Group_free(&group_world);
33
          MPI_Group_free(&subgroup);
34
           MPI_Finalize();
35
           return 0;
36
        }
37
38
            Library Example #1
     7.5.5
39
     The main program:
40
41
        int main(int argc, char *argv[])
42
        {
43
           int done = 0;
44
           user_lib_t *libh_a, *libh_b;
45
           void *dataset1, *dataset2;
46
           . . .
47
          MPI_Init(&argc, &argv);
48
```

```
1
      . . .
                                                                                            2
     init_user_lib(MPI_COMM_WORLD, &libh_a);
                                                                                            3
     init_user_lib(MPI_COMM_WORLD, &libh_b);
                                                                                            4
      . . .
     user_start_op(libh_a, dataset1);
                                                                                            5
                                                                                            6
     user_start_op(libh_b, dataset2);
                                                                                            7
      . . .
     while(!done)
     {
                                                                                            9
                                                                                           10
         /* work */
                                                                                           11
         . . .
         MPI_Reduce(..., MPI_COMM_WORLD);
                                                                                           12
                                                                                           13
         . . .
                                                                                           14
         /* see if done */
                                                                                           15
         . . .
                                                                                           16
     }
                                                                                           17
     user_end_op(libh_a);
                                                                                           18
     user_end_op(libh_b);
                                                                                           19
                                                                                           20
     uninit_user_lib(libh_a);
                                                                                           21
     uninit_user_lib(libh_b);
     MPI_Finalize();
                                                                                           22
                                                                                           23
     return 0;
                                                                                           ^{24}
   }
                                                                                           25
The user library initialization code:
                                                                                           26
                                                                                           27
   void init_user_lib(MPI_Comm comm, user_lib_t **handle)
                                                                                           28
   {
                                                                                           29
     user_lib_t *save;
                                                                                           30
                                                                                           ^{31}
     user_lib_initsave(&save); /* local */
                                                                                           32
     MPI_Comm_dup(comm, &(save->comm));
                                                                                           33
                                                                                           34
     /* other inits
                                                                                           35
      . . .
                                                                                           36
                                                                                           37
     *handle = save;
                                                                                           38
   }
                                                                                           39
User start-up code:
                                                                                           40
                                                                                           41
   void user_start_op(user_lib_t *handle, void *data)
                                                                                           42
   {
                                                                                           43
     MPI_Irecv( ..., handle->comm, &(handle->irecv_handle) );
                                                                                           44
     MPI_Isend( ..., handle->comm, &(handle->isend_handle) );
                                                                                           45
   }
                                                                                           46
                                                                                           47
```

User communication clean-up code:

```
1
        void user_end_op(user_lib_t *handle)
\mathbf{2}
        {
3
          MPI_Status status;
4
          MPI_Wait(&handle->isend_handle, &status);
5
          MPI_Wait(&handle->irecv_handle, &status);
6
        }
7
     User object clean-up code:
8
9
        void uninit_user_lib(user_lib_t *handle)
10
        {
11
          MPI_Comm_free(&(handle->comm));
12
          free(handle);
13
        }
14
15
           Library Example \#2
     7.5.6
16
17
     The main program:
18
        int main(int argc, char *argv[])
19
        ſ
20
          int ma, mb;
21
          MPI_Group group_world, group_a, group_b;
22
          MPI_Comm comm_a, comm_b;
23
24
          static int list_a[] = \{0, 1\};
25
     #if defined(EXAMPLE_2B) || defined(EXAMPLE_2C)
26
          static int list_b[] = {0, 2,3};
27
     #else/* EXAMPLE_2A */
28
          static int list_b[] = {0, 2};
29
     #endif
30
          int size_list_a = sizeof(list_a)/sizeof(int);
31
          int size_list_b = sizeof(list_b)/sizeof(int);
32
33
34
           . . .
          MPI_Init(&argc, &argv);
35
          MPI_Comm_group(MPI_COMM_WORLD, &group_world);
36
37
          MPI_Group_incl(group_world, size_list_a, list_a, &group_a);
38
          MPI_Group_incl(group_world, size_list_b, list_b, &group_b);
39
40
          MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
41
          MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);
42
43
          if(comm_a != MPI_COMM_NULL)
44
             MPI_Comm_rank(comm_a, &ma);
45
          if(comm_b != MPI_COMM_NULL)
46
             MPI_Comm_rank(comm_b, &mb);
47
48
```

```
if(comm_a != MPI_COMM_NULL)
        lib_call(comm_a);
     if(comm_b != MPI_COMM_NULL)
     ſ
       lib_call(comm_b);
       lib_call(comm_b);
     }
     if(comm_a != MPI_COMM_NULL)
       MPI_Comm_free(&comm_a);
     if(comm_b != MPI_COMM_NULL)
       MPI_Comm_free(&comm_b);
     MPI_Group_free(&group_a);
     MPI_Group_free(&group_b);
     MPI_Group_free(&group_world);
     MPI_Finalize();
     return 0;
   }
The library:
                                                                                   21
   void lib_call(MPI_Comm comm)
   ſ
     int me, done = 0;
     MPI_Status status;
     MPI_Comm_rank(comm, &me);
     if(me == 0)
        while(!done)
        {
           MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
           . . .
     else
                                                                                   34
     {
       /* work */
       MPI_Send(..., 0, ARBITRARY_TAG, comm);
     }
#ifdef EXAMPLE_2C
     /* include (resp, exclude) for safety (resp, no safety): */
     MPI_Barrier(comm);
#endif
   }
```

The above example is really three examples, depending on whether or not one includes rank 45463 in list_b, and whether or not a synchronize is included in lib_call. This example illustrates 47that, despite contexts, subsequent calls to lib_call with the same context need not be safe 48 from one another (colloquially, "back-masking"). Safety is realized if the MPI_Barrier is

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5 6

9 10

11

12

13

1415

16

17

18

19

20

22

23

 24

25

26

27

28

29

30

 31

32 33

35

36

37 38

39

40

41

42

43

added. What this demonstrates is that libraries have to be written carefully, even with
 contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from
 back-masking.

⁴ Algorithms like "reduce" and "allreduce" have strong enough source selectivity prop-⁵ erties so that they are inherently okay (no back-masking), provided that MPI provides basic ⁶ guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root ⁷ or different roots (see [64]). Here we rely on two guarantees of MPI: pairwise ordering of ⁸ messages between processes in the same context, and source selectivity — deleting either ⁹ feature removes the guarantee that back-masking cannot be required.

Algorithms that try to do non-deterministic broadcasts or other calls that include wildcard operations will not generally have the good properties of the deterministic implementations of "reduce," "allreduce," and "broadcast." Such algorithms would have to utilize the monotonically increasing tags (within a communicator scope) to keep things straight.

All of the foregoing is a supposition of "collective calls" implemented with point-topoint operations. MPI implementations may or may not implement collective calls using point-to-point operations. These algorithms are used to illustrate the issues of correctness and safety, independent of how MPI implements its collective calls. See also Section 7.9.

7.6 Inter-Communication

This section introduces the concept of inter-communication and describes the portions of MPI that support it. It describes support for writing programs that contain user-level servers.

All communication described thus far has involved communication between processes that are members of the same group. This type of communication is called "intra-communication" and the communicator used is called an "intra-communicator," as we have noted earlier in the chapter.

In modular and multi-disciplinary applications, different process groups execute distinct 29modules and processes within different modules communicate with one another in a pipeline 30 or a more general module graph. In these applications, the most natural way for a process 31 to specify a target process is by the rank of the target process within the target group. In 32 applications that contain internal user-level servers, each server may be a process group that 33 provides services to one or more clients, and each client may be a process group that uses the 34services of one or more servers. It is again most natural to specify the target process by rank 35 within the target group in these applications. This type of communication is called "inter 36 -communication" and the communicator used is called an "inter-communicator," as 37 introduced earlier. 38

An inter-communication is a point-to-point communication between processes in different groups. The group containing a process that initiates an inter-communication operation is called the "local group," that is, the sender in a send and the receiver in a receive. The group containing the target process is called the "remote group," that is, the receiver in a send and the sender in a receive. As in intra-communication, the target process is specified using a (communicator, rank) pair. Unlike intra-communication, the rank is relative to a second, remote group.

All inter-communicator constructors are blocking except for MPI_COMM_IDUP and require that the local and remote groups be disjoint.

47 48

19

20 21

22

23

Advice to users. The groups must be disjoint for several reasons. Primarily, this

is the intent of the intercommunicators — to provide a communicator for communication between disjoint groups. This is reflected in the definition of MPI_INTERCOMM_MERGE, which allows the user to control the ranking of the processes in the created intracommunicator; this ranking makes little sense if the groups are not disjoint. In addition, the natural extension of collective operations to intercommunicators makes the most sense when the groups are disjoint. (*End of advice to users.*)

Here is a summary of the properties of inter-communication and inter-communicators:

- The syntax of point-to-point and collective communication is the same for both interand intra-communication. The same communicator can be used both for send and for receive operations.
- A target process is addressed by its rank in the remote group, both for sends and for receives.
- Communications using an inter-communicator are guaranteed not to conflict with any communications that use a different communicator.
- A communicator will provide either intra- or inter-communication, never both.

The routine MPI_COMM_TEST_INTER may be used to determine if a communicator is an inter- or intra-communicator. Inter-communicators can be used as arguments to some of the other communicator access routines. Inter-communicators cannot be used as input to some of the constructor routines for intra-communicators (for instance, MPI_CART_CREATE).

Advice to implementors. For the purpose of point-to-point communication, communicators can be represented in each process by a tuple consisting of:

group

send_context

receive_context

source

For inter-communicators, *group* describes the remote group, and *source* is the rank of the process in the local group. For intra-communicators, *group* is the communicator group (remote=local), *source* is the rank of the process in this group, and *send context* and *receive context* are identical. A group can be represented by a rank-to-absolute-address translation table.

The inter-communicator cannot be discussed sensibly without considering processes in both the local and remote groups. Imagine a process \mathbf{P} in group \mathcal{P} , which has an intercommunicator $\mathbf{C}_{\mathcal{P}}$, and a process \mathbf{Q} in group \mathcal{Q} , which has an inter-communicator $\mathbf{C}_{\mathcal{Q}}$. Then

- $C_{\mathcal{P}}$.group describes the group \mathcal{Q} and $C_{\mathcal{Q}}$.group describes the group \mathcal{P} .
- $C_{\mathcal{P}}$.send_context = $C_{\mathcal{Q}}$.receive_context and the context is unique in \mathcal{Q} ; $C_{\mathcal{P}}$.receive_context = $C_{\mathcal{Q}}$.send_context and this context is unique in \mathcal{P} .
- $C_{\mathcal{P}}$.source is rank of **P** in \mathcal{P} and $C_{\mathcal{Q}}$.source is rank of **Q** in \mathcal{Q} .

2

3

4

5 6

7

8

9 10

11

12

13

14 15

16

17

18 19

20

21

22

23

24 25

26

27 28

29

30 31

32

33 34

35

36

37

38

39

40

41

42

43 44

45

46

47

1 Assume that \mathbf{P} sends a message to \mathbf{Q} using the inter-communicator. Then \mathbf{P} uses $\mathbf{2}$ the group table to find the absolute address of Q; source and send_context are 3 appended to the message. 4 Assume that \mathbf{Q} posts a receive with an explicit source argument using the inter-5communicator. Then **Q** matches **receive_context** to the message context and source 6 argument to the message source. 7 The same algorithm is appropriate for intra-communicators as well. 8 9 In order to support inter-communicator accessors and constructors, it is necessary to 10 supplement this model with additional structures, that store information about the 11 local communication group, and additional safe contexts. (End of advice to imple*mentors.*) 1213 147.6.1 Inter-Communicator Accessors 151617 MPI_COMM_TEST_INTER(comm, flag) 18 IN comm communicator (handle) 19 OUT true if comm is an inter-communicator (logical) 20flag 2122C binding 23int MPI_Comm_test_inter(MPI_Comm comm, int *flag) 24 Fortran 2008 binding 25MPI_Comm_test_inter(comm, flag, ierror) 26TYPE(MPI_Comm), INTENT(IN) :: comm 27LOGICAL, INTENT(OUT) :: flag 28INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 30 Fortran binding 31 MPI_COMM_TEST_INTER(COMM, FLAG, IERROR) 32 INTEGER COMM, IERROR 33 LOGICAL FLAG 34 This local routine allows the calling process to determine if a communicator is an inter-35 communicator or an intra-communicator. It returns true if it is an inter-communicator, 36 otherwise false. 37 When an inter-communicator is used as an input argument to the communicator ac-38 cessors described above under intra-communication, the following table describes behavior. 39 40 MPI_COMM_SIZE returns the size of the local group. 41 MPI_COMM_GROUP returns the local group. 42MPI_COMM_RANK returns the rank in the local group 43 44 45Table 7.1: MPI COMM * Function Behavior (in Inter-Communication Mode) 46 47Furthermore, the operation MPI_COMM_COMPARE is valid for inter-communicators. Both 48communicators must be either intra- or inter-communicators, or else MPI_UNEQUAL results.

1

48

		articular it is possible for MPI SIMILAR to result	2	
MPI_CONGRUENT or MPI_SIMILAR. In particular, it is possible for MPI_SIMILAR to result ² because either the local or remote groups were similar but not identical. ³			3	
The following accessors provide consistent access to the remote group of an inter-			4	
	communicator. The following are all local operations.			
		-	6	
		X .	7	
MPI_COM	M_REMOTE_SIZE(comm, size)	8	
IN	comm	inter-communicator (handle)	9	
OUT	size	number of processes in the remote group of comm	10 11	
		(integer)		
			12	
C binding	<u>o</u> .		13	
	Comm_remote_size(MPI_Comm	comm, int *size)	14 15	
			16	
	2008 binding		17	
	remote_size(comm, size, i MPI_Comm), INTENT(IN) ::		18	
	ER, INTENT(OUT) :: size	Comm	19	
	ER, OPTIONAL, INTENT(OUT)	:: jerror	20	
			21	
Fortran k	3		22	
	REMOTE_SIZE(COMM, SIZE, 1	ERROR)	23	
INTEGER COMM, SIZE, IERROR			24	
			25	
NOL COMMA DEMOTE CROUD(26	
MPI_COMM_REMOTE_GROUP(comm, group)			27	
IN	comm	inter-communicator (handle)	28 29	
OUT	group	remote group corresponding to comm (handle)	29 30	
			31	
C binding	g		32	
int MPI_C	comm_remote_group(MPI_Comm	n comm, MPI_Group *group)	33	
Fortron 2	2008 binding		34	
	remote_group(comm, group,	ierror)	35	
	MPI_Comm), INTENT(IN) ::		36	
	MPI_Group), INTENT(OUT) :		37	
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
			39	
Fortran b	5		40 41	
	MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)			
			42	
			43 44	
	-	both the local and remote groups of an inter-	44 45	
communicator is important, so this function, as well as MPI_COMM_REMOTE_SIZE			46	
have	have been provided. (<i>End of rationale.</i>)			

Both corresponding local and remote groups must compare correctly to get the results

¹ 7.6.2 Inter-Communicator Operations

 $^2_{_3}$ This section introduces five blocking inter-communicator operations.

⁴ MPI_INTERCOMM_CREATE is used to bind two intra-communicators into an inter-com-⁵ municator; the function MPI_INTERCOMM_CREATE_FROM_GROUPS constructs an inter-⁶ communicator from two previously defined disjoint groups; the function

MPI_INTERCOMM_MERGE creates an intra-communicator by merging the local and remote
 groups of an inter-communicator. The functions MPI_COMM_DUP and MPI_COMM_FREE,
 introduced previously, duplicate and free an inter-communicator, respectively.

Overlap of local and remote groups that are bound into an inter-communicator is prohibited. If there is overlap, then the program is erroneous and is likely to deadlock. (If a process is multithreaded, and MPI calls block only a thread, rather than a process, then "dual membership" can be supported. It is then the user's responsibility to make sure that calls on behalf of the two "roles" of a process are executed by two independent threads.)

The function MPI_INTERCOMM_CREATE can be used to create an inter-communicator from two existing intra-communicators, in the following situation: At least one selected member from each group (the "group leader") has the ability to communicate with the selected member from the other group; that is, a "peer" communicator exists to which both leaders belong, and each leader knows the rank of the other leader in this peer communicator. Furthermore, members of each group know the rank of their leader.

Construction of an inter-communicator from two intra-communicators requires separate collective operations in the local group and in the remote group, as well as a point-to-point communication between a process in the local group and a process in the remote group.

When using the World Model, the MPI_COMM_WORLD communicator (or preferably a dedicated duplicate thereof) can be this peer communicator. For applications that have used the Sessions Model, spawn, or join it may be necessary to first create an intracommunicator to be used as peer.

The application topology functions described in Chapter 8 do not apply to intercommunicators. Users that require this capability should utilize

MPI_INTERCOMM_MERGE to build an intra-communicator, then apply the graph or cartesian topology capabilities to that intra-communicator, creating an appropriate topologyoriented intra-communicator. Alternatively, it may be reasonable to devise one's own application topology mechanisms for this case, without loss of generality.

MPI_INTE	RCOMM_CREATE(local_comr newintercomm)	m, local_leader, peer_comm, remote_leader, tag,	12	
IN	local_comm	local intra-communicator (handle)	3	
IN	local_leader	rank of local group leader in local_comm (integer)	4 5	
IN	peer_comm	"peer" communicator; significant only at the local_leader (handle)	6 7	
IN	remote_leader	rank of remote group leader in peer_comm; significant only at the local_leader (integer)	8 9 10	
IN	tag	tag (integer)	11	
OUT	newintercomm	new inter-communicator (handle)	12	
C bindin int MPI_1	Intercomm_create(MPI_Comm	<pre>local_comm, int local_leader, int remote_leader, int tag,</pre>	13 14 15 16 17	
	2008 binding	local_leader, peer_comm, remote_leader,	18 19 20	
	tag, newintercomm, i		20	
	(MPI_Comm), INTENT(IN) ::	-	22	
INTEGER, INTENT(IN) :: local_leader, remote_leader, tag				
TYPE(MPI_Comm), INTENT(OUT) :: newintercomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror			24	
) Terror	25 26	
Fortran binding MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG, NEWINTERCOMM, IERROR)			27 28 29	
INTEC	GER LOCAL_COMM, LOCAL_LEAN NEWINTERCOMM, IERRO	DER, PEER_COMM, REMOTE_LEADER, TAG, R	29 30 31	
This call creates an inter-communicator. It is collective over the union of the local and			32	
		provide identical local_comm and	33	
		p. Wildcards are not permitted for remote_leader,	34	
local_leade	er, and tag.		35	
			36 37	
			38	
			39	
			42	
			43 44	
			45 46	
			47	
48				

```
1
     MPI_INTERCOMM_CREATE_FROM_GROUPS(local_group, local_leader, remote_group,
\mathbf{2}
                     remote_leader, stringtag, info, errhandler, newintercomm)
3
       IN
                 local_group
                                             local group (handle)
4
       IN
                 local_leader
                                             rank of local group leader in local_group (integer)
5
6
       IN
                                             remote group, significant only at local_leader (handle)
                 remote_group
7
                 remote_leader
       IN
                                             rank of remote group leader in remote_group,
8
                                             significant only at local_leader (integer)
9
                 stringtag
                                             unique idenitifier for this operation (string)
       IN
10
11
                 info
       IN
                                             info object (handle)
12
       IN
                 errhandler
                                             error handler to be attached to new
13
                                             inter-communicator (handle)
14
       OUT
                 newintercomm
                                             new inter-communicator (handle)
15
16
17
     C binding
     int MPI_Intercomm_create_from_groups(MPI_Group local_group,
18
                     int local_leader, MPI_Group remote_group, int remote_leader,
19
                    const char *stringtag, MPI_Info info,
20
                    MPI_Errhandler errhandler, MPI_Comm *newintercomm)
21
22
     Fortran 2008 binding
23
     MPI_Intercomm_create_from_groups(local_group, local_leader, remote_group,
24
                    remote_leader, stringtag, info, errhandler, newintercomm,
25
                     ierror)
26
          TYPE(MPI_Group), INTENT(IN) :: local_group, remote_group
27
          INTEGER, INTENT(IN) :: local_leader, remote_leader
28
          CHARACTER(LEN=*), INTENT(IN) :: stringtag
29
          TYPE(MPI_Info), INTENT(IN) :: info
30
          TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
31
          TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     Fortran binding
     MPI_INTERCOMM_CREATE_FROM_GROUPS(LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP,
35
                    REMOTE_LEADER, STRINGTAG, INFO, ERRHANDLER, NEWINTERCOMM,
36
37
                     IERROR)
          INTEGER LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP, REMOTE_LEADER, INFO,
38
                     ERRHANDLER, NEWINTERCOMM, IERROR
39
40
          CHARACTER*(*) STRINGTAG
41
     This call creates an inter-communicator. Unlike MPI_INTERCOMM_CREATE, this function
42
     uses as input previously defined, disjoint local and remote groups. The calling MPI process
43
     must be a member of the local group. The call is collective over the union of the local
44
     and remote groups. All involved MPI processes shall provide an identical value for the
45
     stringtag argument. Within each group, all MPI processes shall provide identical
46
     local_group, local_leader arguments. Wildcards are not permitted for the
47
     remote_leader or local_leader arguments. The stringtag argument serves the same purpose
48
```

as the stringtag used in the MPI_COMM_CREATE_FROM_GROUP function; it differentiates concurrent calls in a multithreaded environment. The stringtag shall not exceed MPI_MAX_FROM_GROUP_STRINGTAG characters in length. For C, this includes space for a null terminating character. In the event that MPI_GROUP_EMPTY is supplied as the local_group or remote_group or both, then the call is a local operation and MPI_COMM_NULL is returned as the newintercomm.

MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)

IN	intercomm	inter-communicator (handle)
IN	high	ordering of the local and remote groups in the new intra-communicator (logical)
OUT	newintracomm	new intra-communicator (handle)

C binding

int	MPI_Intercomm_merge(MPI_Comm	intercomm, int	; high,
	MPI_Comm *newintra	comm)	

Fortran 2008 binding

MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)
TYPE(MPI_Comm), INTENT(IN) :: intercomm
LOGICAL, INTENT(IN) :: high
TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
INTEGER INTERCOMM, NEWINTRACOMM, IERROR
LOGICAL HIGH

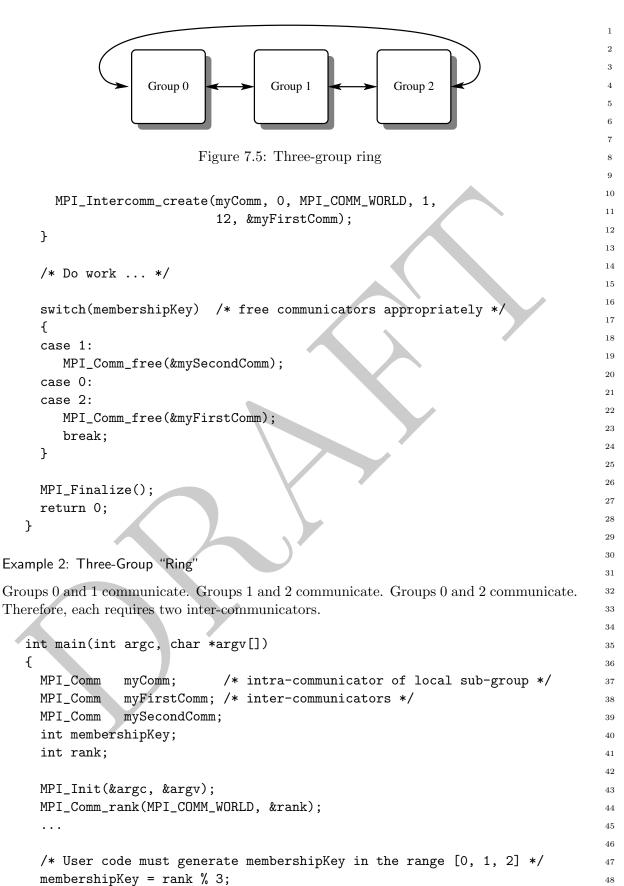
This function creates an intra-communicator from the union of the two groups that are associated with intercomm. All processes should provide the same high value within each of the two groups. If processes in one group provided the value high = false and processes in the other group provided the value high = true then the union orders the "low" group before the "high" group. If all processes provided the same high argument then the order of the union is arbitrary. This call is blocking and collective within the union of the two groups.

The error handler on the new intercommunicator in each process is inherited from the communicator that contributes the local group. Note that this can result in different processes in the same communicator having different error handlers.

Advice to implementors. The implementation of MPI_INTERCOMM_MERGE, MPI_COMM_FREE, and MPI_COMM_DUP are similar to the implementation of MPI_INTERCOMM_CREATE, except that contexts private to the input inter-communicator are used for communication between group leaders rather than contexts inside a bridge communicator. (*End of advice to implementors.*)

Unofficial Draft for Comment Only

```
1
2
3
                      Group 0
                                           Group 1
                                                                Group 2
5
6
7
                               Figure 7.4: Three-group pipeline
8
9
            Inter-Communication Examples
     7.6.3
10
11
     Example 1: Three-Group "Pipeline"
12
     Groups 0 and 1 communicate. Groups 1 and 2 communicate. Therefore, group 0 requires
13
     one inter-communicator, group 1 requires two inter-communicators, and group 2 requires 1
14
     inter-communicator.
15
16
        int main(int argc, char *argv[])
17
        {
18
                                      /* intra-communicator of local sub-group */
          MPI_Comm
                       myComm;
19
          MPI_Comm
                       myFirstComm;
                                      /* inter-communicator */
20
          MPI_Comm
                       mySecondComm; /* second inter-communicator (group 1 only) */
21
           int membershipKey;
22
           int rank;
23
24
           MPI_Init(&argc, &argv);
25
           MPI_Comm_rank(MPI_COMM_WORLD, &rank);
26
27
           /* User code must generate membershipKey in the range [0, 1, 2] */
28
           membershipKey = rank % 3;
29
30
           /* Build intra-communicator for local sub-group */
31
           MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
32
33
           /* Build inter-communicators.
                                             Tags are hard-coded. */
34
           if (membershipKey == 0)
35
                                   /* Group 0 communicates with group 1. */
           {
36
             MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
37
                                    1, &myFirstComm);
38
           }
39
           else if (membershipKey == 1)
40
                           /* Group 1 communicates with groups 0 and 2. */
           Ł
41
             MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
42
                                    1, &myFirstComm);
43
             MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
44
                                    12, &mySecondComm);
45
           }
46
           else if (membershipKey == 2)
47
           {
                                   /* Group 2 communicates with group 1. */
48
```



```
1
2
          /* Build intra-communicator for local sub-group */
3
          MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
4
5
          /* Build inter-communicators. Tags are hard-coded. */
6
          if (membershipKey == 0)
7
          {
                         /* Group 0 communicates with groups 1 and 2. */
8
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
9
                                   1, &myFirstComm);
10
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
11
                                   2, &mySecondComm);
12
          }
13
          else if (membershipKey == 1)
14
                                                                     */
          {
                     /* Group 1 communicates with groups 0 and 2.
15
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
16
                                   1, &myFirstComm);
17
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
18
                                   12, &mySecondComm);
19
          }
20
          else if (membershipKey == 2)
21
                    /* Group 2 communicates with groups 0 and 1. */
          {
22
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
23
                                   2, &myFirstComm);
24
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
25
                                   12, &mySecondComm);
26
          }
27
28
          /* Do some work ...
29
          /* Then free communicators before terminating... */
30
31
          MPI_Comm_free(&myFirstComm);
32
          MPI_Comm_free(&mySecondComm);
33
          MPI_Comm_free(&myComm);
34
          MPI_Finalize();
35
          return 0;
36
        }
37
```

7.7 Caching

38

39 40

41

42

43

44 45

46

47

48

MPI provides a "caching" facility that allows an application to attach arbitrary pieces of information, called **attributes**, to three kinds of MPI objects, communicators, windows, and datatypes. More precisely, the caching facility allows a portable library to do the following:

- pass information between calls by associating it with an MPI intra- or inter-communicator, window, or datatype,
- quickly retrieve that information, and

Unofficial Draft for Comment Only

• be guaranteed that out-of-date information is never retrieved, even if the object is freed and its handle subsequently reused by MPI.

The caching capabilities, in some form, are required by built-in MPI routines such as collective communication and application topology. Defining an interface to these capabilities as part of the MPI standard is valuable because it permits routines like collective communication and application topologies to be implemented as portable code, and also because it makes MPI more extensible by allowing user-written routines to use standard MPI calling sequences.

Advice to users. The communicator MPI_COMM_SELF is a suitable choice for posting process-local attributes, via this attribute-caching mechanism. (*End of advice to* users.)

Rationale. In one extreme one can allow caching on all opaque handles. The other extreme is to only allow it on communicators. Caching has a cost associated with it and should only be allowed when it is clearly needed and the increased cost is modest. This is the reason that windows and datatypes were added but not other handles. (*End of rationale.*)

One difficulty is the potential for size differences between Fortran integers and C pointers. For this reason, the Fortran versions of these routines use integers of kind MPI_ADDRESS_KIND.

Advice to implementors. High-quality implementations should raise an error when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL is used with an object of the wrong type with a call to MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each keyval, information on the type of the associated user function. (*End of advice to implementors.*)

7.7.1 Functionality

Attributes can be attached to communicators, windows, and datatypes. Attributes are local to the process and specific to the communicator to which they are attached. Attributes are not propagated by MPI from one communicator to another except when the communicator is duplicated using MPI_COMM_DUP or MPI_COMM_IDUP (and even then the application must give specific permission through callback functions for the attribute to be copied).

Advice to users. Attributes in C are of type void*. Typically, such an attribute will be a pointer to a structure that contains further information, or a handle to an MPI object. In Fortran, attributes are of type INTEGER. Such attribute can be a handle to an MPI object, or just an integer-valued attribute. (End of advice to users.)

Advice to implementors. Attributes are scalar values, equal in size to, or larger than a C-language pointer. Attributes can always hold an MPI handle. (*End of advice to implementors.*)

The caching interface defined here requires that attributes be stored by MPI opaquely within a communicator, window, or datatype. Accessor functions include the following:

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

 24

 $46 \\ 47$

1 • obtain a key value (used to identify an attribute); the user specifies "callback" func- $\mathbf{2}$ tions by which MPI informs the application when the communicator is destroyed or 3 copied. 4 • store and retrieve the value of an attribute; 56 Advice to implementors. Caching and callback functions are only called synchronously, 7 in response to explicit application requests. This avoids problems that result from re-8 peated crossings between user and system space. (This synchronous calling rule is a 9 general property of MPI.) 10 The choice of key values is under control of MPI. This allows MPI to optimize its 11 implementation of attribute sets. It also avoids conflict between independent modules 12caching information on the same communicators. 13 14A much smaller interface, consisting of just a callback facility, would allow the entire 15caching facility to be implemented by portable code. However, with the minimal call-16back interface, some form of table searching is implied by the need to handle arbitrary 17 communicators. In contrast, the more complete interface defined here permits rapid 18 access to attributes through the use of pointers in communicators (to find the attribute 19 table) and cleverly chosen key values (to retrieve individual attributes). In light of the 20efficiency "hit" inherent in the minimal interface, the more complete interface defined 21here is seen to be superior. (End of advice to implementors.) 22 23MPI provides the following services related to caching. They are all process local. 24 257.7.2 Communicators 26Functions for caching on communicators are: 272829MPI_COMM_CREATE_KEYVAL(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval, 30 extra_state) 31 IN comm_copy_attr_fn copy callback function for comm_keyval (function) 32 33 IN comm_delete_attr_fn delete callback function for comm_keyval (function) 34 OUT comm_keyval key value for future access (integer) 35 IN extra_state extra state for callback function 36 37 38 C binding 39 int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn, 40 MPI_Comm_delete_attr_function *comm_delete_attr_fn, 41 int *comm_keyval, void *extra_state) 42Fortran 2008 binding 43 MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval, 44 extra_state, ierror) 45PROCEDURE(MPI_Comm_copy_attr_function), INTENT(IN) :: comm_copy_attr_fn 46 PROCEDURE(MPI_Comm_delete_attr_function), INTENT(IN) :: 47comm_delete_attr_fn 48

Unofficial Draft for Comment Only

```
1
    INTEGER, INTENT(OUT) :: comm_keyval
                                                                                      2
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
                                                                                      6
              EXTRA_STATE, IERROR)
    EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
    INTEGER COMM_KEYVAL, IERROR
                                                                                      9
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                     10
                                                                                     11
    Generates a new attribute key. Keys are locally unique in a process, and opaque to
user, though they are explicitly stored in integers. Once allocated, the key value can be
                                                                                     12
                                                                                     13
used to associate attributes and access them on any locally defined communicator.
                                                                                     14
The C callback functions are:
                                                                                     15
typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
                                                                                     16
              void *extra_state, void *attribute_val_in,
                                                                                     17
              void *attribute_val_out, int *flag);
                                                                                     18
and
                                                                                     19
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
                                                                                     20
              void *attribute_val, void *extra_state);
                                                                                     21
                                                                                     22
which are the same as the MPI-1.1 calls but with a new name. The old names are deprecated.
                                                                                     23
With the mpi_f08 module, the Fortran callback functions are:
                                                                                     ^{24}
ABSTRACT INTERFACE
                                                                                     25
  SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
                                                                                     26
               attribute_val_in, attribute_val_out, flag, ierror)
                                                                                     27
    TYPE(MPI_Comm) :: oldcomm
                                                                                     28
    INTEGER :: comm_keyval, ierror
                                                                                     29
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                     30
               attribute_val_out
                                                                                     31
    LOGICAL :: flag
                                                                                     32
and
                                                                                     33
ABSTRACT INTERFACE
                                                                                     34
 SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
                                                                                     35
               attribute_val, extra_state, ierror)
                                                                                     36
    TYPE(MPI_Comm) :: comm
                                                                                     37
    INTEGER :: comm_keyval, ierror
                                                                                     38
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                     39
                                                                                     40
With the mpi module and mpif.h, the Fortran callback functions are:
                                                                                     41
SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
                                                                                     42
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                     43
    INTEGER OLDCOMM, COMM_KEYVAL, IERROR
                                                                                     44
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                                                     45
               ATTRIBUTE_VAL_OUT
                                                                                     46
    LOGICAL FLAG
                                                                                     47
                                                                                     48
```

and

1	SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
2	EXTRA_STATE, IERROR)
3	INTEGER COMM, COMM_KEYVAL, IERROR
4	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
5	
6	The comm_copy_attr_fn function is invoked when a communicator is duplicated by
7	MPI_COMM_DUP or MPI_COMM_IDUP. comm_copy_attr_fn should be of type
8	MPI_Comm_copy_attr_function. The copy callback function is invoked for each key value in
9	oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its
10	corresponding attribute. If it returns $flag = 0$ or .FALSE., then the attribute is deleted in
11	the duplicated communicator. Otherwise ($flag = 1$ or .TRUE.), the new attribute value is
12	set to the value returned in attribute_val_out. The function returns MPI_SUCCESS on success
13	and an error code on failure (in which case MPI_COMM_DUP or MPI_COMM_IDUP will
14	fail).
15	The argument comm_copy_attr_fn may be specified as MPI_COMM_NULL_COPY_FN
16	or MPI_COMM_DUP_FN from either C or Fortran. MPI_COMM_NULL_COPY_FN is a function that does not him other than naturning flag. O on FAISE (does not him on what has
17	function that does nothing other than returning $flag = 0$ or .FALSE. (depending on whether the learned mith a C on Fortran hinding to MPL COMMA (DEATE KEN)(AL) and
18	the keyval was created with a C or Fortran binding to MPI_COMM_CREATE_KEYVAL) and MPI_SUCCESS. MPI_COMM_DUP_FN is a simple-minded copy function that sets flag = 1 or
19	.TRUE., returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS.
20	These replace the MPI-1 predefined callbacks MPI_NULL_COPY_FN and MPI_DUP_FN,
21	whose use is deprecated.
22	whose use is deprecated.
23	Advice to users. Even though both formal arguments attribute_val_in and
24	attribute_val_out are of type void*, their usage differs. The C copy function is passed
25	by MPI in attribute_val_in the value of the attribute, and in attribute_val_out the
26	address of the attribute, so as to allow the function to return the (new) attribute
27	value. The use of type void* for both is to avoid messy type casts.
28	A valid copy function is one that completely duplicates the information by making
29	a full duplicate copy of the data structures implied by an attribute; another might
30	just make another reference to that data structure, while using a reference-count
31 32	mechanism. Other types of attributes might not copy at all (they might be specific
33	to oldcomm only). (End of advice to users.)
34	
35	Advice to implementors. A C interface should be assumed for copy and delete
36	functions associated with key values created in C; a Fortran calling interface should
37	be assumed for key values created in Fortran. (End of advice to implementors.)
38	
39	Analogous to comm_copy_attr_fn is a callback deletion function, defined as follows.
40	The comm_delete_attr_fn function is invoked when a communicator is deleted by
41	MPI_COMM_FREE or when a call is made explicitly to MPI_COMM_DELETE_ATTR.
42	comm_delete_attr_fn should be of type MPI_Comm_delete_attr_function.
43	This function is called by MPI_COMM_FREE, MPI_COMM_DELETE_ATTR, and
44	MPI_COMM_SET_ATTR to do whatever is needed to remove an attribute. The function
45	returns MPI_SUCCESS on success and an error code on failure (in which case
46	MPI_COMM_FREE will fail).
47	The argument comm_delete_attr_fn may be specified as
48	MPI_COMM_NULL_DELETE_FN from either C or Fortran.

MPI_COMM_NULL_DELETE_FN is a function that does nothing, other than returning 1 MPI_SUCCESS. MPI_COMM_NULL_DELETE_FN replaces MPI_NULL_DELETE_FN, whose 2 3 use is deprecated. 4 If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_COMM_FREE), 56 is erroneous. 7 The special key value MPI_KEYVAL_INVALID is never returned by MPI_COMM_CREATE_KEYVAL. Therefore, it can be used for static initialization of key 8 9 values. 10 The predefined Fortran functions Advice to implementors. 11 MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and 12MPI_COMM_NULL_DELETE_FN are defined in the mpi module (and mpif.h) and 13the mpi_f08 module with the same name, but with different interfaces. Each function 14can coexist twice with the same name in the same MPI library, one routine as an 15implicit interface outside of the mpi module, i.e., declared as EXTERNAL, and the other 16routine within mpi_f08 declared with CONTAINS. These routines have different link 17names, which are also different to the link names used for the routines used in C. 18 (End of advice to implementors.) 1920Callbacks, including the predefined Fortran functions Advice to users. 21MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and 22 MPI_COMM_NULL_DELETE_FN should not be passed from one application routine 23that uses the mpi_f08 module to another application routine that uses the mpi module 24 or mpif.h, and vice versa; see also the advice to users on page 787. (End of advice to 25users.) 262728 29 MPI_COMM_FREE_KEYVAL(comm_keyval) 30 comm_keyval INOUT key value (integer) 31 32 C binding 33 int MPI_Comm_free_keyval(int *comm_keyval) 34 35Fortran 2008 binding 36 MPI_Comm_free_keyval(comm_keyval, ierror) 37 INTEGER, INTENT(INOUT) :: comm_keyval 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 Fortran binding 40 41 MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR) 42INTEGER COMM_KEYVAL, IERROR 43 Frees an extant attribute key. This function sets the value of keyval to 44MPI_KEYVAL_INVALID. Note that it is not erroneous to free an attribute key that is in use, 45because the actual free does not transpire until after all references (in other communicators 46on the process) to the key have been freed. These references need to be explicitly freed by the

program, either via calls to MPI_COMM_DELETE_ATTR that free one attribute instance,

47

```
1
     or by calls to MPI_COMM_FREE that free all attribute instances associated with the freed
\mathbf{2}
     communicator.
3
4
     MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)
5
6
       INOUT
                 comm
                                              communicator to which attribute will be attached
7
                                              (handle)
8
       IN
                 comm_keyval
                                              key value (integer)
9
       IN
                 attribute_val
                                              attribute value
10
11
     C binding
12
     int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)
13
14
     Fortran 2008 binding
15
     MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)
16
          TYPE(MPI_Comm), INTENT(IN) :: comm
17
          INTEGER, INTENT(IN) :: comm_keyval
18
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
19
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
     Fortran binding
21
     MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
22
23
          INTEGER COMM, COMM_KEYVAL, IERROR
^{24}
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
25
          This function stores the stipulated attribute value attribute_val for subsequent retrieval
26
     by MPI_COMM_GET_ATTR. If the value is already present, then the outcome is as if
27
     MPI_COMM_DELETE_ATTR was first called to delete the previous value (and the callback
28
     function comm_delete_attr_fn was executed), and a new value was next stored. The call
29
     is erroneous if there is no key with value keyval; in particular MPI_KEYVAL_INVALID is an
30
     erroneous key value. The call will fail if the comm_delete_attr_fn function returned an error
^{31}
     code other than MPI_SUCCESS.
32
33
34
     MPI_COMM_GET_ATTR(comm, comm_keyval, attribute_val, flag)
35
       IN
                 comm
                                              communicator to which the attribute is attached
36
                                              (handle)
37
       IN
                                              key value (integer)
38
                 comm_keyval
39
       OUT
                 attribute_val
                                              attribute value, unless flag = false
40
       OUT
                 flag
                                              false if no attribute is associated with the key
41
                                              (logical)
42
43
     C binding
44
     int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
45
                     int *flag)
46
47
     Fortran 2008 binding
48
```

<pre>MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)</pre>
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: comm_keyval
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
LOGICAL, INTENT(OUT) :: flag
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
INTEGER COMM, COMM_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
LOGICAL FLAG
Deteiner etteilete och belev The cellic organistic there is a here

Retrieves attribute value by key. The call is erroneous if there is no key with value keyval. On the other hand, the call is correct if the key value exists, but no attribute is attached on comm for that key; in such case, the call returns flag = false. In particular MPI_KEYVAL_INVALID is an erroneous key value.

Advice to users. The call to MPI_Comm_set_attr passes in attribute_val the value of the attribute; the call to MPI_Comm_get_attr passes in attribute_val the address of the location where the attribute value is to be returned. Thus, if the attribute value itself is a pointer of type void*, then the actual attribute_val parameter to MPI_Comm_set_attr will be of type void* and the actual attribute_val parameter to MPI_Comm_get_attr will be of type void**. (End of advice to users.)

Rationale. The use of a formal parameter attribute_val of type void* (rather than void**) avoids the messy type casting that would be needed if the attribute value is declared with a type other than void*. (*End of rationale.*)

		20
MPI_COMM_DELETE_ATTR(comm, comm_keyval)		
		31
	communicator from which the attribute is deleted	32
	(handle)	33
IN comm_keyval	key value (integer)	34
		35
C binding		36
int MPI_Comm_delete_attr(MPI_Comm_c	comm. int.comm.keyval)	37
Int in I_comm_delete_det (in I_comm comm, into comm_keyval)		
Fortran 2008 binding		
<pre>MPI_Comm_delete_attr(comm, comm_keyval, ierror)</pre>		
TYPE(MPI_Comm), INTENT(IN) :: comm		
INTEGER, INTENT(IN) :: comm_keyval		
INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
Fortran binding		
MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR) INTEGER COMM, COMM_KEYVAL, IERROR		
INTEGER COMM, COMM_RETVAL, IERR	.016	47

```
1
         Delete attribute from cache by key. This function invokes the attribute delete function
\mathbf{2}
     comm_delete_attr_fn specified when the keyval was created. The call will fail if the
3
     comm_delete_attr_fn function returns an error code other than MPI_SUCCESS.
4
         Whenever a communicator is replicated using the function MPI_COMM_DUP or
     MPI_COMM_IDUP, all call-back copy functions for attributes that are currently set are
\mathbf{5}
6
     invoked (in arbitrary order). Whenever a communicator is deleted using the function
7
     MPI_COMM_FREE all callback delete functions for attributes that are currently set are
8
     invoked.
9
10
     7.7.3 Windows
11
     The functions for caching on windows are:
12
13
14
     MPI_WIN_CREATE_KEYVAL(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
15
                    extra_state)
16
       IN
                win_copy_attr_fn
                                             copy callback function for win_keyval (function)
17
18
                win_delete_attr_fn
                                             delete callback function for win_keyval (function)
       IN
19
       OUT
                win_keyval
                                             key value for future access (integer)
20
       IN
                 extra_state
                                             extra state for callback function
21
22
23
     C binding
     int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,
^{24}
25
                    MPI_Win_delete_attr_function *win_delete_attr_fn,
26
                    int *win_keyval, void *extra_state)
27
     Fortran 2008 binding
28
     MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
29
                    extra_state, ierror)
30
         PROCEDURE(MPI_Win_copy_attr_function), INTENT(IN) :: win_copy_attr_fn
31
         PROCEDURE(MPI_Win_delete_attr_function), INTENT(IN) ::
32
                     win_delete_attr_fn
33
          INTEGER, INTENT(OUT) :: win_keyval
34
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     Fortran binding
38
     MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
39
                    EXTRA_STATE, IERROR)
40
         EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
41
          INTEGER WIN_KEYVAL, IERROR
42
          INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
43
         The argument win_copy_attr_fn may be specified as MPI_WIN_NULL_COPY_FN or
44
     MPI_WIN_DUP_FN from either C or Fortran. MPI_WIN_NULL_COPY_FN is a function
45
     that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_WIN_DUP_FN is
46
     a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in
47
```

```
The argument win_delete_attr_fn may be specified as MPI_WIN_NULL_DELETE_FN
                                                                                      1
                                                                                      \mathbf{2}
from either C or Fortran. MPI_WIN_NULL_DELETE_FN is a function that does nothing,
other than returning MPI_SUCCESS.
The C callback functions are:
                                                                                      4
typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
                                                                                      5
                                                                                      6
              void *extra_state, void *attribute_val_in,
              void *attribute_val_out, int *flag);
                                                                                      7
and
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
                                                                                      10
              void *attribute_val, void *extra_state);
                                                                                      11
With the mpi_f08 module, the Fortran callback functions are:
                                                                                      12
                                                                                      13
ABSTRACT INTERFACE
                                                                                      14
  SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
                                                                                      15
               attribute_val_in, attribute_val_out, flag, ierror)
                                                                                      16
    TYPE(MPI_Win) :: oldwin
                                                                                      17
    INTEGER :: win_keyval, ierror
                                                                                      18
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                      19
               attribute_val_out
                                                                                      20
    LOGICAL :: flag
                                                                                      21
and
                                                                                      22
ABSTRACT INTERFACE
                                                                                      23
  SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
                                                                                      24
               extra_state, ierror)
                                                                                      25
    TYPE(MPI_Win) :: win
                                                                                      26
    INTEGER :: win_keyval, ierror
                                                                                      27
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                      28
                                                                                      29
With the mpi module and mpif.h, the Fortran callback functions are:
                                                                                      30
SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
                                                                                      31
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                      32
    INTEGER OLDWIN, WIN_KEYVAL, IERROR
                                                                                      33
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                                                      34
               ATTRIBUTE_VAL_OUT
                                                                                      35
    LOGICAL FLAG
                                                                                      36
and
                                                                                      37
SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
                                                                                      38
              EXTRA_STATE, IERROR)
                                                                                      39
    INTEGER WIN, WIN_KEYVAL, IERROR
                                                                                      40
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                      41
                                                                                      42
    If an attribute copy function or attribute delete function returns other than
                                                                                      43
MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_WIN_FREE), is
                                                                                      44
erroneous.
                                                                                      45
                                                                                      46
                                                                                      47
```

```
1
     MPI_WIN_FREE_KEYVAL(win_keyval)
\mathbf{2}
       INOUT
                 win_keyval
                                              key value (integer)
3
4
     C binding
5
     int MPI_Win_free_keyval(int *win_keyval)
6
\overline{7}
     Fortran 2008 binding
8
     MPI_Win_free_keyval(win_keyval, ierror)
9
          INTEGER, INTENT(INOUT) :: win_keyval
10
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
     Fortran binding
12
     MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
13
          INTEGER WIN_KEYVAL, IERROR
14
15
16
     MPI_WIN_SET_ATTR(win, win_keyval, attribute_val)
17
18
                                              window to which attribute will be attached (handle)
       INOUT
                 win
19
       IN
                 win_keyval
                                              key value (integer)
20
21
       IN
                 attribute_val
                                              attribute value
22
23
     C binding
^{24}
     int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
25
     Fortran 2008 binding
26
     MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
27
          TYPE(MPI_Win), INTENT(IN) :: win
28
          INTEGER, INTENT(IN) :: win_keyval
29
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
30
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
     Fortran binding
33
     MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
34
          INTEGER WIN, WIN_KEYVAL, IERROR
35
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
36
37
38
     MPI_WIN_GET_ATTR(win, win_keyval, attribute_val, flag)
39
       IN
                 win
                                              window to which the attribute is attached (handle)
40
41
                 win_keyval
       IN
                                              key value (integer)
42
       OUT
                 attribute_val
                                              attribute value, unless flag = false
43
       OUT
                 flag
                                              false if no attribute is associated with the key
44
                                              (logical)
45
46
47
     C binding
48
```

7.7. CACHING

<pre>int MPI_Win_get_attr(MPI_Win win, i</pre>	nt win_keyval, void *attribute_val,	1 2
Fortran 2008 binding		3
MPI_Win_get_attr(win, win_keyval, a	ttribute val. flag. ierror)	4
TYPE(MPI_Win), INTENT(IN) :: wi		5
INTEGER, INTENT(IN) :: win_keyv		6 7
INTEGER(KIND=MPI_ADDRESS_KIND),		8
LOGICAL, INTENT(OUT) :: flag		9
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	10
Fortran binding		11
MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, A	TTRIBUTE VAL FLAG (FRROR)	12
INTEGER WIN, WIN_KEYVAL, IERROR		13
INTEGER(KIND=MPI_ADDRESS_KIND)		14
LOGICAL FLAG		15
		16
		17
MPI_WIN_DELETE_ATTR(win, win_keyva		18
、 -	, .	19
	window from which the attribute is deleted (handle)	20
IN win_keyval	key value (integer)	21
		22
C binding		23 24
int MPI_Win_delete_attr(MPI_Win win	, int win_keyval)	24 25
Fortran 2008 binding		26
MPI_Win_delete_attr(win, win_keyval	, ierror)	27
TYPE(MPI_Win), INTENT(IN) :: wi		28
INTEGER, INTENT(IN) :: win_keyv	al	29
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	30
Fortran binding		31
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL	. TERBOR)	32
INTEGER WIN, WIN_KEYVAL, IERROR		33
,,,		34
		35
7.7.4 Datatypes		36
The new functions for caching on datatyp	es are:	37
		38
		39
Ť		40
		41 42
		42
		43
		45
		46
		47

```
1
     MPI_TYPE_CREATE_KEYVAL(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
\mathbf{2}
                    extra_state)
3
       IN
                type_copy_attr_fn
                                           copy callback function for type_keyval (function)
4
       IN
                type_delete_attr_fn
                                           delete callback function for type_keyval (function)
5
6
       OUT
                type_keyval
                                           key value for future access (integer)
7
       IN
                extra_state
                                           extra state for callback function
8
9
     C binding
10
     int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
11
                    MPI_Type_delete_attr_function *type_delete_attr_fn,
12
                    int *type_keyval, void *extra_state)
13
14
     Fortran 2008 binding
15
     MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
16
                    extra_state, ierror)
17
         PROCEDURE(MPI_Type_copy_attr_function), INTENT(IN) :: type_copy_attr_fn
18
         PROCEDURE(MPI_Type_delete_attr_function), INTENT(IN) ::
19
                    type_delete_attr_fn
20
         INTEGER, INTENT(OUT) :: type_keyval
21
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     Fortran binding
24
     MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
25
                    EXTRA_STATE, IERROR)
26
         EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
27
         INTEGER TYPE_KEYVAL, IERROR
28
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
29
30
         The argument type_copy_attr_fn may be specified as MPI_TYPE_NULL_COPY_FN or
31
     MPI_TYPE_DUP_FN from either C or Fortran. MPI_TYPE_NULL_COPY_FN is a function
32
     that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_TYPE_DUP_FN
33
     is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in
34
     attribute_val_out, and returns MPI_SUCCESS.
35
         The argument type_delete_attr_fn may be specified as MPI_TYPE_NULL_DELETE_FN
36
     from either C or Fortran. MPI_TYPE_NULL_DELETE_FN is a function that does nothing,
37
     other than returning MPI_SUCCESS.
38
     The C callback functions are:
39
     typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
40
                    int type_keyval, void *extra_state, void *attribute_val_in,
41
                    void *attribute_val_out, int *flag);
42
     and
43
     typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype,
44
                    int type_keyval, void *attribute_val, void *extra_state);
45
46
     With the mpi_f08 module, the Fortran callback functions are:
47
     ABSTRACT INTERFACE
48
```

```
1
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
                                                                                     2
               attribute_val_in, attribute_val_out, flag, ierror)
                                                                                     3
    TYPE(MPI_Datatype) :: oldtype
    INTEGER :: type_keyval, ierror
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                     5
               attribute_val_out
                                                                                     6
    LOGICAL :: flag
and
                                                                                     9
ABSTRACT INTERFACE
                                                                                     10
  SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
                                                                                     11
               attribute_val, extra_state, ierror)
                                                                                     12
    TYPE(MPI_Datatype) :: datatype
                                                                                     13
    INTEGER :: type_keyval, ierror
                                                                                     14
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                     15
                                                                                     16
With the mpi module and mpif.h, the Fortran callback functions are:
                                                                                     17
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
                                                                                     18
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                     19
    INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                                                     20
                                                                                     21
               ATTRIBUTE_VAL_OUT
                                                                                     22
    LOGICAL FLAG
                                                                                     23
and
                                                                                     24
SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
                                                                                     25
              EXTRA_STATE, IERROR)
                                                                                     26
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
                                                                                     27
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                     28
                                                                                     29
    If an attribute copy function or attribute delete function returns other than
                                                                                     30
MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_TYPE_FREE),
                                                                                     31
is erroneous.
                                                                                     32
                                                                                     33
MPI_TYPE_FREE_KEYVAL(type_keyval)
                                                                                     34
                                                                                     35
 INOUT type_keyval
                                     key value (integer)
                                                                                     36
                                                                                     37
C binding
                                                                                     38
int MPI_Type_free_keyval(int *type_keyval)
                                                                                     39
Fortran 2008 binding
                                                                                     40
                                                                                     41
MPI_Type_free_keyval(type_keyval, ierror)
    INTEGER, INTENT(INOUT) :: type_keyval
                                                                                     42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     43
                                                                                     44
Fortran binding
                                                                                     45
MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
                                                                                     46
    INTEGER TYPE_KEYVAL, IERROR
                                                                                     47
                                                                                     48
```

```
1
     MPI_TYPE_SET_ATTR(datatype, type_keyval, attribute_val)
2
       INOUT
                datatype
                                            datatype to which attribute will be attached (handle)
3
                type_keyval
       IN
                                            key value (integer)
4
5
                 attribute_val
       IN
                                            attribute value
6
7
     C binding
8
     int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,
9
                    void *attribute_val)
10
     Fortran 2008 binding
11
     MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
12
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
13
         INTEGER, INTENT(IN) :: type_keyval
14
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     Fortran binding
18
     MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
19
         INTEGER DATATYPE, TYPE_KEYVAL, IERROR
20
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
21
22
23
     MPI_TYPE_GET_ATTR(datatype, type_keyval, attribute_val, flag)
24
                                            datatype to which the attribute is attached (handle)
       IN
                 datatype
25
26
       IN
                 type_keyval
                                            key value (integer)
27
       OUT
                 attribute_val
                                            attribute value, unless flag = false
28
       OUT
                flag
                                            false if no attribute is associated with the key
29
                                            (logical)
30
31
32
     C binding
33
     int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval,
34
                    void *attribute_val, int *flag)
35
     Fortran 2008 binding
36
     MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
37
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
         INTEGER, INTENT(IN) :: type_keyval
39
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
40
         LOGICAL, INTENT(OUT) :: flag
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     Fortran binding
44
     MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
45
         INTEGER DATATYPE, TYPE_KEYVAL, IERROR
46
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
47
         LOGICAL FLAG
48
```

INOUT	datatype	datatype from which the attribute is deleted (handle)
IN	type_keyval	key value (integer)
		
binding		
.nt MPI_I	ype_delete_attr(MF	YI_Datatype datatype, int type_keyval)
Fortran 2	2008 binding	
	-	pe, type_keyval, ierror)
		ENT(IN) :: datatype
	ER, INTENT(IN) ::	
INTEC	ER, OPTIONAL, INTE	NT(UUT) :: lerror
Fortran h	oinding	
IPI_TYPE_	DELETE_ATTR(DATATY	PE, TYPE_KEYVAL, IERROR)
INTEC	ER DATATYPE, TYPE_	KEYVAL, IERROR
7.7.5 Eri	or Class for Invalid K	evval
	s for attributes are sys $()$	
	-	Only such values can be passed to the functions that use
-		In order to signal that an erroneous key value has been , there is a new MPI error class: MPI_ERR_KEYVAL. It can
		T, MPI_ATTR_GET, MPI_ATTR_DELETE,
	/AL_FREE,	
	<pre>{}_DELETE_ATTR,</pre>	
	<pre>{}_SET_ATTR,</pre>	
ΛΡΙ_{ΧΧΧ	(}_GET_ATTR,	
		IPI_COMM_DUP, MPI_COMM_IDUP,
		d MPI_COMM_FREE. The last four are included because
eyval is a	n argument to the cop	y and delete functions for attributes.
7.7.6 At	ributes Example	7
Advi	ce to users. This	example shows how to write a collective communication
		g to be more efficient after the first call. (End of advice to
user.		
1. 1		
ĩ	for this module's	
static	: int gop_key = MP1	_NEIVAL_INVALID;
typede	ef struct	
typede {	JI SULUCU	
	; ref_count;	/* reference count */
/*	other stuff, whate	ver erse we want */
	other stuff, whate .stuff_type;	AAT ETZE ME Maint */

```
1
        void Efficient_Collective_Op(MPI_Comm comm, ...)
\mathbf{2}
        {
3
          gop_stuff_type *gop_stuff;
4
          MPI_Group
                           group;
5
          int
                           foundflag;
6
7
          MPI_Comm_group(comm, &group);
8
9
          if (gop_key == MPI_KEYVAL_INVALID) /* get a key on first call ever */
10
          {
11
            if ( ! MPI_Comm_create_keyval(gop_stuff_copier,
12
                                       gop_stuff_destructor,
13
                                       &gop_key, (void *)0)) {
14
            /* get the key while assigning its copy and delete callback
15
               behavior. */
16
            } else
17
                MPI_Abort(comm, 99);
18
          }
19
20
          MPI_Comm_get_attr(comm, gop_key, &gop_stuff, &foundflag);
21
          if (foundflag)
22
          { /* This module has executed in this group before.
23
                We will use the cached information */
24
          }
25
          else
26
          { /* This is a group that we have not yet cached anything in.
27
                We will now do so.
28
            */
29
30
            /* First, allocate storage for the stuff we want,
31
                and initialize the reference count */
32
33
            gop_stuff = (gop_stuff_type *) malloc(sizeof(gop_stuff_type));
34
            if (gop_stuff == NULL) { /* abort on out-of-memory error */ }
35
36
            gop_stuff->ref_count = 1;
37
38
            /* Second, fill in *gop_stuff with whatever we want.
39
                This part isn't shown here */
40
41
            /* Third, store gop_stuff as the attribute value */
42
            MPI_Comm_set_attr(comm, gop_key, gop_stuff);
43
          }
44
          /* Then, in any case, use contents of *gop_stuff
45
             to do the global op ... */
46
        }
47
48
        /* The following routine is called by MPI when a group is freed */
```

Unofficial Draft for Comment Only

```
int gop_stuff_destructor(MPI_Comm comm, int keyval, void *gop_stuffP,
                         void *extra)
ł
  gop_stuff_type *gop_stuff = (gop_stuff_type *)gop_stuffP;
  if (keyval != gop_key) { /* abort -- programming error */ }
  /* The group's being freed removes one reference to gop_stuff */
  gop_stuff->ref_count -= 1;
  /* If no references remain, then free the storage */
  if (gop_stuff->ref_count == 0) {
    free((void *)gop_stuff);
  }
  return MPI_SUCCESS;
}
/* The following routine is called by MPI when a group is copied */
int gop_stuff_copier(MPI_Comm comm, int keyval, void *extra,
               void *gop_stuff_inP, void *gop_stuff_outP, int *flag)
ł
  gop_stuff_type *gop_stuff_in = (gop_stuff_type *)gop_stuff_inP;
  gop_stuff_type **gop_stuff_out = (gop_stuff_type **)gop_stuff_outP;
  if (keyval != gop_key) { /* abort -- programming error */ }
  /* The new group adds one reference to this gop_stuff */
  gop_stuff_in->ref_count += 1;
  *gop_stuff_out = gop_stuff_in;
  return MPI_SUCCESS;
}
```

7.8 Naming Objects

There are many occasions on which it would be useful to allow a user to associate a printable identifier with an MPI communicator, window, or datatype, for instance error reporting, debugging, and profiling. The names attached to opaque objects do not propagate when the object is duplicated or copied by MPI routines. For communicators this can be achieved using the following two functions.

			40
MPI_COMM_SET_NAME(comm, comm_name)			41
INOUT	comm	communicator whose identifier is to be set (handle)	42 43
IN	comm_name	the character string which is remembered as the	44
		name (string)	45
			46
C binding			47
int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)			48

 $\mathbf{2}$

10 11

12

13 14

15

16

17 18

19

20

21

22

23 24

> 25 26

27

28 29

30 31 32

33 34

35

36

37

38

1		008 binding	
2		<pre>set_name(comm, comm_name,</pre>	
3	TYPE(MPI_Comm), INTENT(IN) :: comm		
4		CTER(LEN=*), INTENT(IN) :	
5 6	INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror
7	Fortran b	inding	
8	MPI_COMM_	SET_NAME(COMM, COMM_NAME,	IERROR)
9	INTEG	ER COMM, IERROR	
10	CHARA	CTER*(*) COMM_NAME	
11	MPI C	OMM SET NAME allows a us	ser to associate a name string with a communicator.
12			MPI_COMM_SET_NAME will be saved inside the
13	MPI library	y (so it can be freed by the ca	ller immediately after the call, or allocated on the
14	stack). Lea	ding spaces in name are signif	ficant but trailing ones are not.
15	MPI_C	OMM_SET_NAME is a local	(non-collective) operation, which only affects the
16	name of the	e communicator as seen in the	process which made the MPI_COMM_SET_NAME
17	call. There	is no requirement that the sa	me (or any) name be assigned to a communicator
18	in every pre-	ocess where it exists.	
19	1 days	a to warma Since MPL COM	M_SET_NAME is provided to help debug code, it
20			to a communicator in all of the processes where it
21		, to avoid confusion. (End of	-
22 23	CAIDID		
23		-	be stored is limited to the value of
25	$MPI_MAX_OBJECT_NAME$ in Fortran and $MPI_MAX_OBJECT_NAME-1$ in C to allow for the		
26	null terminator. Attempts to put names longer than this will result in truncation of the		
27	name. MPL	_MAX_OBJECT_NAME must ha	ave a value of at least 64.
28	Advia	e to users. Under circumstar	nces of store exhaustion an attempt to put a name
29	of any length could fail, therefore the value of MPI_MAX_OBJECT_NAME should be		
30	viewe	d only as a strict upper bound	d on the name length, not a guarantee that setting
31	name	s of less than this length will	always succeed. (End of advice to users.)
32	Advic	e to implementors Implement	ntations which pre-allocate a fixed size space for a
33		-	llocation as the value of MPI_MAX_OBJECT_NAME.
34		Ū.	ace for the name from the heap should still define
35 36	MPI_MAX_OBJECT_NAME to be a relatively small value, since the user has to allocate		
37	space for a string of up to this size when calling MPI_COMM_GET_NAME. (End of		
38		e to implementors.)	
39			
40			
41	MPI_COM	M_GET_NAME(comm, comm_	name, resultlen)
42	IN	comm	communicator whose name is to be returned (handle)
43 44	OUT	comm_name	the name previously stored on the communicator, or
44 45			an empty string if no such name exists (string)
46	OUT	resultlen	length of returned name (integer)
47			0
48	C binding		

int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)

Fortran 2008 binding

```
MPI_Comm_get_name(comm, comm_name, resultlen, ierror)
   TYPE(MPI_Comm), INTENT(IN) :: comm
   CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
   INTEGER, INTENT(OUT) :: resultlen
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
INTEGER COMM, RESULTLEN, IERROR
CHARACTER*(*) COMM_NAME
```

MPI_COMM_GET_NAME returns the last name which has previously been associated with the given communicator. The name may be set and retrieved from any language. The same name will be returned independent of the language used. name should be allocated so that it can hold a resulting string of length MPI_MAX_OBJECT_NAME characters. MPI_COMM_GET_NAME returns a copy of the set name in name.

In C, a null character is additionally stored at name[resultlen]. The value of resultlen cannot be larger than MPI_MAX_OBJECT_NAME-1. In Fortran, name is padded on the right with blank characters. The value of resultlen cannot be larger than MPI_MAX_OBJECT_NAME.

If the user has not associated a name with a communicator, or an error occurs, MPI_COMM_GET_NAME will return an empty string (all spaces in Fortran, "" in C). The three predefined communicators will have predefined names associated with them. Thus, the names of MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator returned by MPI_COMM_GET_PARENT (if not MPI_COMM_NULL) will have the default of MPI_COMM_WORLD, MPI_COMM_SELF, and MPI_COMM_PARENT. The fact that the system may have chosen to give a default name to a communicator does not prevent the user from setting a name on the same communicator; doing this removes the old name and assigns the new one.

Rationale. We provide separate functions for setting and getting the name of a communicator, rather than simply providing a predefined attribute key for the following reasons:

- It is not, in general, possible to store a string as an attribute from Fortran.
- It is not easy to set up the delete function for a string attribute unless it is known to have been allocated from the heap.
- To make the attribute key useful additional code to call strdup is necessary. If this is not standardized then users have to write it. This is extra unneeded work which we can easily eliminate.
- The Fortran binding is not trivial to write (it will depend on details of the Fortran compilation system), and will not be portable. Therefore it should be in the library rather than in user code.

(End of rationale.)

1

2

5

6

10

11

12 13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31 32

33

34

35 36

37

38

39

40

41

42

43 44

 $45 \\ 46$

```
1
           Advice to users. The above definition means that it is safe simply to print the string
\mathbf{2}
           returned by MPI_COMM_GET_NAME, as it is always a valid string even if there was
3
           no name.
4
           Note that associating a name with a communicator has no effect on the semantics of
5
           an MPI program, and will (necessarily) increase the store requirement of the program,
6
           since the names must be saved. Therefore there is no requirement that users use these
7
           functions to associate names with communicators. However debugging and profiling
8
           MPI applications may be made easier if names are associated with communicators,
9
           since the debugger or profiler should then be able to present information in a less
10
           cryptic manner. (End of advice to users.)
11
12
          The following functions are used for setting and getting names of datatypes. The
13
     constant MPI_MAX_OBJECT_NAME also applies to these names.
14
15
16
     MPI_TYPE_SET_NAME(datatype, type_name)
17
       INOUT
                 datatype
                                              datatype whose identifier is to be set (handle)
18
       IN
                                              the character string which is remembered as the
                 type_name
19
                                              name (string)
20
21
     C binding
22
     int MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)
23
^{24}
     Fortran 2008 binding
25
     MPI_Type_set_name(datatype, type_name, ierror)
26
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
          CHARACTER(LEN=*), INTENT(IN) :: type_name
28
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     Fortran binding
30
     MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
^{31}
          INTEGER DATATYPE, IERROR
32
          CHARACTER*(*) TYPE_NAME
33
34
35
36
     MPI_TYPE_GET_NAME(datatype, type_name, resultlen)
37
       IN
                 datatype
                                              datatype whose name is to be returned (handle)
38
       OUT
                 type_name
                                              the name previously stored on the datatype, or an
39
                                              empty string if no such name exists (string)
40
41
       OUT
                 resultlen
                                              length of returned name (integer)
42
43
     C binding
44
     int MPI_Type_get_name(MPI_Datatype datatype, char *type_name,
45
                     int *resultlen)
46
47
     Fortran 2008 binding
     MPI_Type_get_name(datatype, type_name, resultlen, ierror)
48
```

CHARA INTEG	TYPE(MPI_Datatype), INTENT(IN) :: datatype CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name INTEGER, INTENT(OUT) :: resultlen INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
Fortran binding MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR) INTEGER DATATYPE, RESULTLEN, IERROR CHARACTER*(*) TYPE_NAME						
Named predefined datatypes have the default names of the datatype name. For example, MPI_WCHAR has the default name of "MPI_WCHAR". The following functions are used for setting and getting names of windows. The constant MPI_MAX_OBJECT_NAME also applies to these names.						
MPI_WIN_	SET_NAME(win, win_name)		15 16			
INOUT	win	window whose identifier is to be set (handle)	17			
IN	win_name	the character string which is remembered as the name (string)	18 19 20			
C binding						
•	S Vin_set_name(MPI_Win win,	const char *win_name)	22 23			
Fortran 2	008 binding		24			
	et_name(win, win_name, i	error)	25			
TYPE(MPI_Win), INTENT(IN) :: •	win	26			
	CTER(LEN=*), INTENT(IN)		27 28			
INTEG	ER, OPTIONAL, INTENT(OUT) :: ierror	28 29			
Fortran b	oinding		30			
	ET_NAME(WIN, WIN_NAME, I	ERROR)	31			
	ER WIN, IERROR		32			
CHARA	CTER*(*) WIN_NAME		33			
			34			
	CET NAME(win win name	recultion)	35 36			
	GET_NAME(win, win_name,	,	30			
IN	win	window whose name is to be returned (handle)	38			
OUT	win_name	the name previously stored on the window, or an empty string if no such name exists (string)	39 40			
OUT	resultlen	length of returned name (integer)	41			
			42			
C binding						
<pre>int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)</pre>						
Fortran 2008 binding						
MPI_Win_get_name(win, win_name, resultlen, ierror)						
TYPE(MPI_Win), INTENT(IN) :: win4748						

CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name INTEGER, INTENT(OUT) :: resultlen INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR) INTEGER WIN, RESULTLEN, IERROR CHARACTER*(*) WIN_NAME

7.9 Formalizing the Loosely Synchronous Model

In this section, we make further statements about the loosely synchronous model, with particular attention to intra-communication.

7.9.1 Basic Statements

17When a caller passes a communicator (that contains a context and group) to a callee, that 18 communicator must be free of side effects throughout execution of the subprogram: there 19should be no active operations on that communicator that might involve the process. This 20provides one model in which libraries can be written, and work "safely." For libraries 21so designated, the callee has permission to do whatever communication it likes with the 22communicator, and under the above guarantee knows that no other communications will 23interfere. Since we permit good implementations to create new communicators without 24 synchronization (such as by preallocated contexts on communicators), this does not impose 25a significant overhead.

This form of safety is analogous to other common computer-science usages, such as passing a descriptor of an array to a library routine. The library routine has every right to expect such a descriptor to be valid and modifiable.

29 30

 31

1

2

3

4

5

6

7

8 9 10

11 12

13

14 15

16

7.9.2 Models of Execution

In the loosely synchronous model, transfer of control to a **parallel procedure** is effected by having each executing process invoke the procedure. The invocation is a collective operation: it is executed by all processes in the execution group, and invocations are similarly ordered at all processes. However, the invocation need not be synchronized.

We say that a parallel procedure is *active* in a process if the process belongs to a group that may collectively execute the procedure, and some member of that group is currently executing the procedure code. If a parallel procedure is active in a process, then this process may be receiving messages pertaining to this procedure, even if it does not currently execute the code of this procedure.

 41

⁴² Static Communicator Allocation

This covers the case where, at any point in time, at most one invocation of a parallel procedure can be active at any process, and the group of executing processes is fixed. For example, all invocations of parallel procedures involve all processes, processes are singlethreaded, and there are no recursive invocations.

In such a case, a communicator can be statically allocated to each procedure. The static allocation can be done in a preamble, as part of initialization code. If the parallel procedures can be organized into libraries, so that only one procedure of each library can be concurrently active in each processor, then it is sufficient to allocate one communicator per library.

Dynamic Communicator Allocation

Calls of parallel procedures are well-nested if a new parallel procedure is always invoked in a subset of a group executing the same parallel procedure. Thus, processes that execute the same parallel procedure have the same execution stack.

In such a case, a new communicator needs to be dynamically allocated for each new invocation of a parallel procedure. The allocation is done by the caller. A new communicator can be generated by a call to MPI_COMM_DUP, if the callee execution group is identical to the caller execution group, or by a call to MPI_COMM_SPLIT if the caller execution group is split into several subgroups executing distinct parallel routines. The new communicator is passed as an argument to the invoked routine.

The need for generating a new communicator at each invocation can be alleviated or avoided altogether in some cases: If the execution group is not split, then one can allocate a stack of communicators in a preamble, and next manage the stack in a way that mimics the stack of recursive calls.

One can also take advantage of the well-ordering property of communication to avoid confusing caller and callee communication, even if both use the same communicator. To do so, one needs to abide by the following two rules:

- messages sent before a procedure call (or before a return from the procedure) are also received before the matching call (or return) at the receiving end;
- messages are always selected by source (no use is made of MPI_ANY_SOURCE).

The General Case

In the general case, there may be multiple concurrently active invocations of the same parallel procedure within the same group; invocations may not be well-nested. A new communicator needs to be created for each invocation. It is the user's responsibility to make sure that, should two distinct parallel procedures be invoked concurrently on overlapping sets of processes, communicator creation is properly coordinated. 1

2

3

4

5 6 7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24 25

26

27

28 29 30

31

32

33

34

35



Chapter 8

Process Topologies

8.1 Introduction

This chapter discusses the MPI topology mechanism. A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to intercommunicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

As stated in Chapter 7, a process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement is described by a graph. In this chapter we will refer to this logical process arrangement as the "virtual topology."

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal with machine-independent mapping and communication on virtual process topologies.

Rationale. Though physical mapping is not discussed, the existence of the virtual topology information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [49]. On the other hand, if there is no way for the user to specify the logical process arrangement as a "virtual topology," a random mapping is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [12, 13].

Unofficial Draft for Comment Only

 24

 31

Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with significant benefits for program readability and notational power in message-passing programming. (*End of rationale.*)

8.2 Virtual Topologies

The communication pattern of a set of processes can be represented by a graph. The nodes represent processes, and the edges connect processes that communicate with each other. MPI provides message-passing between any pair of processes in a group. There is no requirement for opening a channel explicitly. Therefore, a "missing link" in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that this connection is neglected in the virtual topology. This strategy implies that the topology gives no convenient way of naming this pathway of communication. Another possible consequence is that an automatic mapping tool (if one exists for the runtime environment) will not take account of this edge when mapping.

16Specifying the virtual topology in terms of a graph is sufficient for all applications. 17However, in many applications the graph structure is regular, and the detailed set-up of the 18 graph would be inconvenient for the user and might be less efficient at run time. A large frac-19tion of all parallel applications use process topologies like rings, two- or higher-dimensional 20grids, or tori. These structures are completely defined by the number of dimensions and 21the numbers of processes in each coordinate direction. Also, the mapping of grids and tori 22is generally an easier problem than that of general graphs. Thus, it is desirable to address 23these cases explicitly. 24

Process coordinates in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a Cartesian structure. This means that, for example, the relation between group rank and coordinates for four processes in a (2×2) grid is as follows.

coord $(0,0)$:	rank 0
coord $(0,1)$:	$\operatorname{rank} 1$
coord $(1,0)$:	$\operatorname{rank} 2$
coord $(1,1)$:	$\operatorname{rank} 3$

8.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 7.

8.4 Overview of the Functions

⁴³ MPI supports three topology types: **Cartesian**, **graph**, and **distributed graph**. The ⁴⁴ function MPI_CART_CREATE is used to create Cartesian topologies, the function

⁴⁶ MPI_GRAPH_CREATE is used to create graph topologies, and the functions

⁴⁰ MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE are used to cre-

 $_{48}$ ate distributed graph topologies. These topology creation functions are collective. As with

354

3 4 5

6 7

8

9

10

11

12

13

14

15

25

26

27

35 36

37

38

39

40 41

42

1

other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The topology creation functions take as input an existing communicator comm_old, which defines the set of processes on which the topology is to be mapped. For MPI_GRAPH_CREATE and MPI_CART_CREATE, all input arguments must have identical values on all processes of the group of comm_old. When calling MPI_GRAPH_CREATE, each process specifies all nodes and edges in the graph. In contrast, the functions MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE are used to specify the graph in a distributed fashion, whereby each process only specifies a subset of the edges in the graph such that the entire graph structure is defined collectively across the set of processes. Therefore the processes provide different values for the arguments specifying the graph. However, all processes must give the same value for reorder and the info argument. In all cases, a new communicator **comm_topol** is created that carries the topological structure as cached information (see Chapter 7). In analogy to function MPI_COMM_CREATE, no cached information propagates from **comm_old** to **comm_topol**.

MPI_CART_CREATE can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an n-dimensional hypercube is an n-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The 20local auxiliary function MPI_DIMS_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

MPI defines functions to query a communicator for topology information. The function MPI_TOPO_TEST is used to query for the type of topology associated with a communicator. Depending on the topology type, different information can be extracted. For a graph topology, the functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET return the values that were specified in the call to MPI_GRAPH_CREATE. Additionally, the functions MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS can be used to obtain the neighbors of an arbitrary node in the graph. For a distributed graph topology, the functions MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS can be used to obtain the neighbors of the calling process. For a Cartesian topology, the functions MPI_CARTDIM_GET and MPI_CART_GET return the values that were specified in the call to MPI_CART_CREATE. Additionally, the functions MPI_CART_RANK and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa. 34The function MPI_CART_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local.

For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group.

The two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP, are, in general, not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 7, they are sufficient to implement all other topology functions. Section 8.5.8 outlines such an implementation.

The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topology associated with the communicator. The nonblocking variants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV,

MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 $\overline{7}$

8 9

10 11

12

13

14

15

16

17

18

19

21

22

23 24

2526

2728

2930

 31

32 33

35

36 37

38

39

40 41

42

43

44

4546

47

12	MPI_INEIGHBOR_ALLTOALLW.					
3 4	8.5 Topology Constructors					
5 6 7	8.5.1 Cartesian Constructor					
8 9	MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)					
10	IN	comm_old	input communicator (handle)			
11	IN	ndims	number of dimensions of Cartesian grid (integer)			
12 13 14	IN	dims	integer array of size ndims specifying the number of processes in each dimension			
15 16	IN	periods	logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension			
17 18	IN	reorder	ranking may be reordered (true) or not (false) (logical)			
19 20	OUT	comm_cart	communicator with new Cartesian topology (handle)			
23 24 25 26 27 28 29 30 31 32	<pre>const int periods[], int reorder, MPI_Comm *comm_cart) Fortran 2008 binding MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm_old INTEGER, INTENT(IN) :: ndims, dims(ndims) LOGICAL, INTENT(IN) :: periods(ndims), reorder TYPE(MPI_Comm), INTENT(OUT) :: comm_cart INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>					
33 34 35 36	Fortran binding MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR) INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR LOGICAL PERIODS(*), REORDER					
37 38 39 40 41 42 43 44 45 46 47 48	MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian topology information is attached. If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes (possibly so as to choose a good embedding of the virtual topology onto the physical machine). If the total size of the Cartesian grid is smaller than the size of the group of comm_old, then some processes are returned MPI_COMM_NULL, in analogy to MPI_COMM_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created. The call is erroneous if it specifies a grid that is larger than the group size or if ndims is negative.					

8.5.2 Cartesian Convenience Function	: MPI_DIMS_CREATE	1
For Cartesian topologies, the function M	PI_DIMS_CREATE helps the user select a balanced	2
· 0 /	direction, depending on the number of processes	3 4
in the group to be balanced and option	nal constraints that can be specified by the user.	5
One use is to partition all the processes	; (the size of MPI_COMM_WORLD's group) into an	6
n-dimensional topology.		7
		8
MPI_DIMS_CREATE(nnodes, ndims, dim	5)	9
· ·		10
	number of nodes in a grid (integer)	11
IN ndims	number of Cartesian dimensions (integer)	12
INOUT dims	integer array of size ndims specifying the number of	13
	nodes in each dimension	14 15
		16
C binding		17
<pre>int MPI_Dims_create(int nnodes, in</pre>	nt ndims, int dims[])	18
Fortran 2008 binding		19
MPI_Dims_create(nnodes, ndims, dim	ns, ierror)	20
INTEGER, INTENT(IN) :: nnodes,	, ndims	21
INTEGER, INTENT(INOUT) :: dims		22
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	23
Fortran binding		24
MPI_DIMS_CREATE(NNODES, NDIMS, DIM	MS, IERROR)	25 26
INTEGER NNODES, NDIMS, DIMS(*)	, IERROR	20
The entries in the array dime are set	to describe a Cartesian grid with ndims dimensions	28
	ions are set to be as close to each other as possible,	29
	m. The caller may further constrain the operation	30
	array dims. If dims[i] is set to a positive number,	31
	of nodes in dimension i; only those entries where	32
dims[i] = 0 are modified by the call.		33
Negative input values of dims[i] are	erroneous. An error will occur if nnodes is not a	34
multiple of	H	35
	dims[i].	36
i,din	$ns[i]{ eq}0$	37
	ill be ordered in non-increasing order. Array dims	38 39
	IPI_CART_CREATE. MPI_DIMS_CREATE is local.	40
If ndims is zero and nnodes is one, $MPI_$	DIMS_CREATE returns MPI_SUCCESS.	41
Example 8.1		42
		43
		44

```
function call
                                                                 dims
1
                   dims
2
                   before call
                                                                 on return
3
                   (0,0)
                                MPI_DIMS_CREATE(6, 2, dims)
                                                                 (3,2)
                                MPI_DIMS_CREATE(7, 2, dims)
4
                   (0,0)
                                                                 (7,1)
                                MPI_DIMS_CREATE(6, 3, dims)
5
                    (0,3,0)
                                                                 (2,3,1)
6
                   (0,3,0)
                                MPI_DIMS_CREATE(7, 3, dims)
                                                                 erroneous call
7
8
            Graph Constructor
     8.5.3
9
10
11
     MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)
12
       IN
                 comm_old
                                             input communicator (handle)
13
14
       IN
                 nnodes
                                             number of nodes in graph (integer)
15
                 index
                                             array of integers describing node degrees (see below)
       IN
16
                 edges
                                             array of integers describing graph edges (see below)
       IN
17
18
       IN
                 reorder
                                             ranking may be reordered (true) or not (false)
19
                                             (logical)
20
       OUT
                 comm_graph
                                             communicator with graph topology added (handle)
21
22
     C binding
23
     int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],
24
                     const int edges[], int reorder, MPI_Comm *comm_graph)
25
26
     Fortran 2008 binding
27
     MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
28
                     ierror)
29
          TYPE(MPI_Comm), INTENT(IN) :: comm_old
30
          INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
31
          LOGICAL, INTENT(IN) :: reorder
32
          TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
33
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     Fortran binding
35
     MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
36
                     IERROR)
37
          INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
38
          LOGICAL REORDER
39
40
          MPI_GRAPH_CREATE returns a handle to a new communicator to which the graph
41
     topology information is attached. If reorder = false then the rank of each process in the
42
     new group is identical to its rank in the old group. Otherwise, the function may reorder the
43
     processes. If the size, nnodes, of the graph is smaller than the size of the group of comm_old,
44
     then some processes are returned MPI_COMM_NULL, in analogy to MPI_CART_CREATE
45
     and MPI_COMM_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI_COMM_NULL
46
     is returned in all processes. The call is erroneous if it specifies a graph that is larger than
47
     the group size of the input communicator.
48
```

The three parameters nnodes, index and edges define the graph structure. nnodes is the number of nodes of the graph. The nodes are numbered from 0 to nnodes-1. The i-th entry of array index stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., nnodes-1 are stored in consecutive locations in array edges. The array edges is a flattened representation of the edge lists. The total number of entries in index is nnodes and the total number of entries in edges is equal to the number of graph edges.

The definitions of the arguments nnodes, index, and edges are illustrated with the following simple example.

Example 8.2 Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

Then, the input arguments are:

$$\begin{array}{ll} \text{nnodes} = & 4 \\ \text{index} = & 2, \, 3, \, 4, \, 6 \\ \text{edges} = & 1, \, 3, \, 0, \, 3, \, 0, \, 2 \end{array}$$

Thus, in C, index[0] is the degree of node zero, and index[i] - index[i-1] is the degree of node i, i=1, ..., nnodes-1; the list of neighbors of node zero is stored in edges[j], for $0 \le j \le index[0] - 1$ and the list of neighbors of node i, i > 0, is stored in edges[j], index[i-1] $\le j \le index[i] - 1$.

In Fortran, index(1) is the degree of node zero, and index(i+1) - index(i) is the degree of node i, i=1, ..., nnodes-1; the list of neighbors of node zero is stored in edges(j), for $1 \le j \le$ index(1) and the list of neighbors of node i, i > 0, is stored in edges(j), index(i)+1 \le j \le index(i+1).

A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be non-symmetric.

Advice to users. Performance implications of using multiple edges or a non-symmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. (*End of advice to users.*)

Advice to implementors. The following topology information is likely to be stored with a communicator:

- Type of topology (Cartesian/graph),
 For a Cartesian topology:
 1. ndims (number of dimensions),
 - 2. dims (numbers of processes per coordinate direction),

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

1	3. periods (periodicity information),
2	4. own_position (own position in grid, could also be computed from rank and
3	dims)
4	
5	• For a graph topology:
6	1. index,
7	2. edges,
8	which are the vectors defining the graph structure.
9	
10	For a graph structure the number of nodes is equal to the number of processes in
11	the group. Therefore, the number of nodes does not have to be stored explicitly.
12	An additional zero entry at the start of array index simplifies access to the topology
13	information. (End of advice to implementors.)
14	
15	8.5.4 Distributed Graph Constructor
16	
17	MPI_GRAPH_CREATE requires that each process passes the full (global) communication
18	graph to the call. This limits the scalability of this constructor. With the distributed graph
19	interface, the communication graph is specified in a fully distributed fashion. Each process
20	specifies only the part of the communication graph of which it is aware. Typically, this
21	could be the set of processes from which the process will eventually receive or get data,

or the set of processes to which the process will send or put data, or some combination of such edges. Two different interfaces can be used to create a distributed graph topology. MPI_DIST_GRAPH_CREATE_ADJACENT creates a distributed graph communicator with each process specifying each of its incoming and outgoing (adjacent) edges in the logical communication graph and thus requires minimal communication during creation.

²⁷ MPI_DIST_GRAPH_CREATE provides full flexibility such that any process can indicate that ²⁸ communication will occur between any pair of processes in the graph.

To provide better possibilities for optimization by the MPI library, the distributed graph constructors permit weighted communication edges and take an info argument that can further influence process reordering or other optimizations performed by the MPI library. For example, hints can be provided on how edge weights are to be interpreted, the quality of the reordering, and/or the time permitted for the MPI library to process the graph.

Unofficial Draft for Comment Only

MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, sourceweights, outdegree, destinations, destweights, info, reorder, comm_dist_graph)

	outdegree, destinations, d	estweights, into, reorder, comm_dist_graph)	
IN	comm_old	input communicator (handle)	3
IN	indegree	size of sources and sourceweights arrays (non-negative integer)	4 5 6
IN	sources	ranks of processes for which the calling process is a destination (array of non-negative integers)	7 8
IN	sourceweights	weights of the edges into the calling process (array of non-negative integers)	9 10 11
IN	outdegree	size of destinations and destweights arrays (non-negative integer)	11 12 13
IN	destinations	ranks of processes for which the calling process is a source (array of non-negative integers)	14 15
IN	destweights	weights of the edges out of the calling process (array of non-negative integers)	16 17 18
IN	info	hints on optimization and interpretation of weights (handle)	19 20
IN	reorder	the ranks may be reordered (true) or not (false) (logical)	21 22
OUT	comm_dist_graph	communicator with distributed graph topology (handle)	23 24
C binding			25 26 27

C binding

int MPI_Dis	t_graph_create_adjacent(MPI_Comm comm_old, int indegree,
	<pre>const int sources[], const int sourceweights[], int outdegree,</pre>
	<pre>const int destinations[], const int destweights[],</pre>
	MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)

Fortran 2008 binding

5	
<pre>MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,</pre>	33
outdegree, destinations, destweights, info, reorder,	34
comm_dist_graph, ierror)	35
TYPE(MPI_Comm), INTENT(IN) :: comm_old	36
<pre>INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*),</pre>	37
<pre>outdegree, destinations(outdegree), destweights(*)</pre>	38
TYPE(MPI_Info), INTENT(IN) :: info	39
LOGICAL, INTENT(IN) :: reorder	40
TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph	41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42
	43
Fortran binding	44
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,	45
OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,	46
COMM_DIST_GRAPH, IERROR)	47

 $\mathbf{2}$

INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR LOGICAL REORDER

5MPI_DIST_GRAPH_CREATE_ADJACENT returns a handle to a new communicator 6 to which the distributed graph topology information is attached. Each process passes all 7 information about its incoming and outgoing edges in the virtual distributed graph topology. 8 The calling processes must ensure that each edge of the graph is described in the source 9 and in the destination process with the same weights. If there are multiple edges for a given 10 (source,dest) pair, then the sequence of the weights of these edges does not matter. The 11 complete communication topology is the combination of all edges shown in the sources arrays 12of all processes in **comm_old**, which must be identical to the combination of all edges shown 13 in the destinations arrays. Source and destination ranks must be process ranks of comm_old. 14This allows a fully distributed specification of the communication graph. Isolated processes 15(i.e., processes with no outgoing or incoming edges, that is, processes that have specified 16indegree and outdegree as zero and thus do not occur as source or destination rank in the 17graph specification) are allowed. 18

The call creates a new communicator comm_dist_graph of distributed graph topology type to which topology information has been attached. The number of processes in comm_dist_graph is identical to the number of processes in comm_old. The call to MPI_DIST_GRAPH_CREATE_ADJACENT is collective.

- Weights are specified as non-negative integers and can be used to influence the process 23remapping strategy and other internal MPI optimizations. For instance, approximate count 24 arguments of later communication calls along specific edges could be used as their edge 25weights. Multiplicity of edges can likewise indicate more intense communication between 26pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 27standard and is left to the implementation. In C or Fortran, an application can supply 28the special value MPL_UNWEIGHTED for the weight array to indicate that all edges have 29 the same (effectively no) weight. It is erroneous to supply MPI_UNWEIGHTED for some 30 but not all processes of comm_old. If the graph is weighted but indegree or outdegree is 31 zero, then MPI_WEIGHTS_EMPTY or any arbitrary array may be passed to sourceweights 32 or destweights respectively. Note that MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are 33 not special weight values; rather they are special values for the total array argument. In 34Fortran, MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are objects like MPI_BOTTOM (not 35 usable for initialization or assignment). See Section 2.5.4. 36
 - Advice to users. In the case of an empty weights array argument passed while constructing a weighted graph, one should not pass NULL because the value of MPI_UNWEIGHTED may be equal to NULL. The value of this argument would then be indistinguishable from MPI_UNWEIGHTED to the implementation. In this case MPI_WEIGHTS_EMPTY should be used instead. (*End of advice to users.*)
 - Advice to implementors. It is recommended that MPI_UNWEIGHTED not be implemented as NULL. (End of advice to implementors.)
- Rationale. To ensure backward compatibility, MPI_UNWEIGHTED may still be imple mented as NULL. See Annex B.4. (End of rationale.)

1

2

3

4

37

38

39

40

41

42 43

44

The meaning of the info and reorder arguments is defined in the description of the following routine.

MPI_DIST_GRAPH_CREATE(comm_old, n,	sources,	degrees,	destinations,	weights,	info,
reorder, comm_dist_graph)					

	= = = = = = = = = = = = = = = = = = = =		0
IN	comm_old	input communicator (handle)	7
IN	n	number of source nodes for which this process	8
		specifies edges (non-negative integer)	9
IN	sources	array containing the \boldsymbol{n} source nodes for which this	10 11
	3041003	process specifies edges (array of non-negative	11
		integers)	13
IN	degrees	array specifying the number of destinations for each	14
IIN	degrees	source node in the source node array (array of	15
		non-negative integers)	16
IN	destinations	destination nodes for the source nodes in the source	17
IIN		node array (array of non-negative integers)	18
IN	weights	weights for source to destination edges (array of	19
IIN	weights	non-negative integers)	20
	:- C-	J J J J J J J J J J J J J J J J J J J	21 22
IN	info	hints on optimization and interpretation of weights (handle)	22
			24
IN	reorder	the ranks may be reordered (true) or not (false)	25
		(logical)	26
OUT	comm_dist_graph	communicator with distributed graph topology	27
		added (handle)	28
~			29
C bindin	-		30
int MPL_		<pre>m comm_old, int n, const int sources[],</pre>	31
		, const int destinations[], , MPI_Info info, int reorder,	32 33
	MPI_Comm *comm_dist		34
		-01 obu)	35
	2008 binding		36
MPI_Dist		n, sources, degrees, destinations, weights,	37
יייעריי		_dist_graph, ierror)	38
	(MPI_Comm), INTENT(IN) ::	<pre>comm_old urces(n), degrees(n), destinations(*),</pre>	39
	weights(*)	nces(n), degrees(n), destinations(*),	40
ТҮРЕ	(MPI_Info), INTENT(IN) ::	info	41
	CAL, INTENT(IN) :: reorde		42
	(MPI_Comm), INTENT(OUT) :		43
	GER, OPTIONAL, INTENT(OUT	. .	44
			$45 \\ 46$
Fortran	5	I, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,	40 47
ומות"ד זיי		_DIST_GRAPH, IERROR)	48

Unofficial Draft for Comment Only

INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*), WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR

LOGICAL REORDER

MPI_DIST_GRAPH_CREATE returns a handle to a new communicator to which the 5distributed graph topology information is attached. Concretely, each process calls the con-6 structor with a set of directed (source.destination) communication edges as described below. 7 Every process passes an array of n source nodes in the sources array. For each source node, a 8 non-negative number of destination nodes is specified in the degrees array. The destination 9 nodes are stored in the corresponding consecutive segment of the destinations array. More 10 precisely, if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the 11 j-th such edge stored in destinations[degrees[0]+ \dots +degrees[i-1]+j]. The weight of this edge 12is stored in weights[degrees[0]+ \ldots +degrees[i-1]+i]. Both the sources and the destinations 13 arrays may contain the same node more than once, and the order in which nodes are listed 14as destinations or sources is not significant. Similarly, different processes may specify edges 15with the same source and destination nodes. Source and destination nodes must be pro-16cess ranks of comm_old. Different processes may specify different numbers of source and 17destination nodes, as well as different source to destination edges. This allows a fully dis-18 tributed specification of the communication graph. Isolated processes (i.e., processes with 19no outgoing or incoming edges, that is, processes that do not occur as source or destination 20node in the graph specification) are allowed. 21

The call creates a new communicator comm_dist_graph of distributed graph topology type to which topology information has been attached. The number of processes in comm_dist_graph is identical to the number of processes in comm_old. The call to MPI_DIST_GRAPH_CREATE is collective.

If reorder = false, all processes will have the same rank in comm_dist_graph as in comm_old. If reorder = true then the MPI library is free to remap to other processes (of comm_old) in order to improve communication on the edges of the communication graph. The weight associated with each edge is a hint to the MPI library about the amount or intensity of communication on that edge, and may be used to compute a "best" reordering.

Weights are specified as non-negative integers and can be used to influence the process 31 remapping strategy and other internal MPI optimizations. For instance, approximate count 32 arguments of later communication calls along specific edges could be used as their edge 33 weights. Multiplicity of edges can likewise indicate more intense communication between 34pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 35 standard and is left to the implementation. In C or Fortran, an application can supply 36 the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the 37 same (effectively no) weight. It is erroneous to supply MPI_UNWEIGHTED for some but not 38 all processes of comm_old. If the graph is weighted but n = 0, then MPI_WEIGHTS_EMPTY 39 or any arbitrary array may be passed to weights. Note that MPI_UNWEIGHTED and 40 MPI_WEIGHTS_EMPTY are not special weight values; rather they are special values for the 41 total array argument. In Fortran, MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are objects 42like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4. 43

Advice to users. In the case of an empty weights array argument passed while
 constructing a weighted graph, one should not pass NULL because the value of
 MPI_UNWEIGHTED may be equal to NULL. The value of this argument would then
 be indistinguishable from MPI_UNWEIGHTED to the implementation.
 MPI_WEIGHTS_EMPTY should be used instead. (End of advice to users.)

Unofficial Draft for Comment Only

1

2

3

Advice to implementors. It is recommended that MPI_UNWEIGHTED not be implemented as NULL. (End of advice to implementors.)

Rationale. To ensure backward compatibility, MPI_UNWEIGHTED may still be implemented as NULL. See Annex B.4. (*End of rationale.*)

The meaning of the weights argument can be influenced by the info argument. Info arguments can be used to guide the mapping; possible options include minimizing the maximum number of edges between processes on different SMP nodes, or minimizing the sum of all such edges. An MPI implementation is not obliged to follow specific hints, and it is valid for an MPI implementation not to do any reordering. An MPI implementation may specify more info key-value pairs. All processes must specify the same set of key-value info pairs.

Advice to implementors. MPI implementations must document any additionally supported key-value info pairs. MPI_INFO_NULL is always valid, and may indicate the default creation of the distributed graph topology to the MPI library.

An implementation does not explicitly need to construct the topology from its distributed parts. However, all processes can construct the full topology from the distributed specification and use this in a call to MPI_GRAPH_CREATE to create the topology. This may serve as a reference implementation of the functionality, and may be acceptable for small communicators. However, a scalable high-quality implementation would save the topology graph in a distributed way. (*End of advice to implementors.*)

Example 8.3 As for Example 8.2, assume there are four processes 0, 1, 2, 3 with the following adjacency matrix and unit edge weights:

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

With MPI_DIST_GRAPH_CREATE, this graph could be constructed in many different ways. One way would be that each process specifies its outgoing edges. The arguments per process would be:

process	n	sources	degrees	destinations	weights
0	1	0	2	$1,\!3$	1,1
1	1	1	1	0	1
2	1	2	1	3	1
3	1	3	2	0,2	1,1

Another way would be to pass the whole graph on process 0, which could be done with the following arguments per process:

process	n	sources	degrees	destinations	weights
0	4	0,1,2,3	2,1,1,2	1,3,0,3,0,2	$1,\!1,\!1,\!1,\!1,\!1,\!1$
1	0	-	-	-	-
2	0	-	-	-	-
3	0	-	-	-	

Unofficial Draft for Comment Only

 $\overline{7}$

In both cases above, the application could supply MPI_UNWEIGHTED instead of explicitly providing identical weights.

MPI_DIST_GRAPH_CREATE_ADJACENT could be used to specify this graph using the following arguments:

[process	indegree	sources	sourceweights	outdegree	destinations	destweights
	0	2	1,3	1,1	2	1,3	1,1
	1	1	0	1	1	0	1
	2	1	3	1	1	3	1
	3	2	$0,\!2$	$1,\!1$	2	0,2	1,1

Example 8.4 A two-dimensional PxQ torus where all processes communicate along the dimensions and along the diagonal edges. This cannot be modeled with Cartesian topologies, but can easily be captured with MPI_DIST_GRAPH_CREATE as shown in the following code. In this example, the communication along the dimensions is twice as heavy as the communication along the diagonals:

18 /*

```
19
     Input:
                 dimensions P, Q
20
     Condition: number of processes equal to P*Q; otherwise only
21
                ranks smaller than P*Q participate
22
     */
23
     int rank, x, y;
^{24}
     int sources[1], degrees[1];
25
     int destinations[8], weights[8];
26
     MPI_Comm comm_dist_graph;
27
28
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
29
30
     /* get x and y dimension */
^{31}
     y=rank/P; x=rank%P;
32
33
     /* get my communication partners along x dimension */
34
     destinations[0] = P*y+(x+1)%P; weights[0] = 2;
     destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;
35
36
37
     /* get my communication partners along y dimension */
38
     destinations[2] = P*((y+1)%Q)+x; weights[2] = 2;
39
     destinations[3] = P*((Q+y-1))(Q)+x; weights[3] = 2;
40
41
     /* get my communication partners along diagonals */
42
     destinations[4] = P*((y+1)%Q)+(x+1)%P; weights[4] = 1;
43
     destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 1;
     destinations[6] = P*((y+1)%Q)+(P+x-1)%P; weights[6] = 1;
44
45
     destinations[7] = P*((Q+y-1)%Q)+(P+x-1)%P; weights[7] = 1;
46
47
     sources[0] = rank;
48
     degrees [0] = 8;
```

1

 $\mathbf{2}$

3

4

MPI_Dist	0 1	RLD, 1, sources, degrees, destinations, I_INFO_NULL, 1, &comm_dist_graph);	1 2 3
8.5.5 To	pology Inquiry Functions		4
-	gy has been defined with one of ked up using inquiry function	f the above functions, then the topology information s. They all are local calls.	5 6 7 8
MPI_TOP	O_TEST(comm, status)		9 10
IN	comm	communicator (handle)	10
OUT	status	topology type of communicator comm (state)	12
C bindin int MPI_1	g fopo_test(MPI_Comm comm, :	int *status)	13 14 15
MPI_Topo TYPE INTE(2008 binding _test(comm, status, ierro: (MPI_Comm), INTENT(IN) :: GER, INTENT(OUT) :: status GER, OPTIONAL, INTENT(OUT)	comm s	16 17 18 19 20 21
Fortran binding MPI_TOPO_TEST(COMM, STATUS, IERROR) INTEGER COMM, STATUS, IERROR			22 23 24 25
communic		eturns the type of topology that is assigned to a ne following:	26 27 28
		graph topology Cartesian topology distributed graph topology no topology	29 30 31 32 33 34
MPI GRA	PHDIMS_GET(comm, nnodes,	nedges)	$35 \\ 36$
IN	comm	communicator for group with graph structure (handle)	37 38
OUT	nnodes	number of nodes in graph (same as number of processes in the group) (integer)	39 40 41
OUT	nedges	number of edges in graph (integer)	41 42 43
C binding int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges) Fortran 2008 binding MPI_Graphdims_get(comm, nnodes, nedges, ierror)			44 45 46 47 48
_			10

```
1
          TYPE(MPI_Comm), INTENT(IN) :: comm
\mathbf{2}
          INTEGER, INTENT(OUT) :: nnodes, nedges
3
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     Fortran binding
5
     MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
6
          INTEGER COMM, NNODES, NEDGES, IERROR
7
8
         Functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET retrieve the graph-topology
9
     information that was associated with a communicator by MPI_GRAPH_CREATE.
10
         The information provided by MPI_GRAPHDIMS_GET can be used to dimension the
11
     vectors index and edges correctly for the following call to MPI_GRAPH_GET.
12
13
     MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges)
14
15
       IN
                 comm
                                             communicator with graph structure (handle)
16
       IN
                 maxindex
                                             length of vector index in the calling program (integer)
17
       IN
                 maxedges
                                             length of vector edges in the calling program (integer)
18
19
       OUT
                 index
                                             array of integers containing the graph structure (for
20
                                             details see the definition of MPI_GRAPH_CREATE)
21
       OUT
                 edges
                                             array of integers containing the graph structure
22
23
     C binding
24
     int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[],
25
                    int edges[])
26
27
     Fortran 2008 binding
28
     MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
29
          TYPE(MPI_Comm), INTENT(IN) :: comm
30
          INTEGER, INTENT(IN) :: maxindex, maxedges
^{31}
          INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
     Fortran binding
34
     MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
35
          INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
36
37
38
     MPI_CARTDIM_GET(comm, ndims)
39
40
       IN
                                             communicator with Cartesian structure (handle)
                 comm
41
       OUT
                 ndims
                                             number of dimensions of the Cartesian structure
42
                                             (integer)
43
44
     C binding
45
     int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
46
47
     Fortran 2008 binding
48
```

```
1
MPI_Cartdim_get(comm, ndims, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                          2
    INTEGER, INTENT(OUT) :: ndims
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          5
Fortran binding
                                                                                          6
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR
                                                                                          9
    The functions MPI_CARTDIM_GET and MPI_CART_GET return the Cartesian topol-
                                                                                          10
ogy information that was associated with a communicator by MPI_CART_CREATE. If comm
                                                                                          11
is associated with a zero-dimensional Cartesian topology, MPI_CARTDIM_GET returns
ndims = 0 and MPI_CART_GET will keep all output arguments unchanged.
                                                                                          12
                                                                                          13
                                                                                          14
MPI_CART_GET(comm, maxdims, dims, periods, coords)
                                                                                          15
                                                                                          16
  IN
                                       communicator with Cartesian structure (handle)
           comm
                                                                                          17
  IN
            maxdims
                                       length of vectors dims, periods, and coords in the
                                                                                          18
                                       calling program (integer)
                                                                                          19
  OUT
           dims
                                        number of processes for each Cartesian dimension
                                                                                          20
                                        (array of integers)
                                                                                          21
                                                                                          22
                                        periodicity (true/false) for each Cartesian dimension
  OUT
            periods
                                                                                          23
                                        (array of logicals)
                                                                                          ^{24}
  OUT
           coords
                                        coordinates of calling process in Cartesian structure
                                                                                          25
                                        (array of integers)
                                                                                          26
                                                                                          27
C binding
                                                                                          28
int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],
                                                                                          29
               int coords[])
                                                                                          30
                                                                                          31
Fortran 2008 binding
                                                                                          32
MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)
                                                                                          33
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                          34
    INTEGER, INTENT(IN) :: maxdims
                                                                                          35
    INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
                                                                                          36
    LOGICAL, INTENT(OUT) :: periods(maxdims)
                                                                                          37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          38
Fortran binding
                                                                                          39
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
                                                                                          40
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
                                                                                          41
    LOGICAL PERIODS(*)
                                                                                          42
                                                                                          43
                                                                                          44
                                                                                          45
                                                                                          46
                                                                                          47
```

```
1
     MPI_CART_RANK(comm, coords, rank)
2
       IN
                  comm
                                               communicator with Cartesian structure (handle)
3
       IN
                 coords
                                               integer array (of size ndims) specifying the Cartesian
4
                                               coordinates of a process
5
6
       OUT
                 rank
                                               rank of specified process (integer)
7
8
      C binding
9
      int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)
10
      Fortran 2008 binding
11
      MPI_Cart_rank(comm, coords, rank, ierror)
12
          TYPE(MPI_Comm), INTENT(IN) :: comm
13
          INTEGER, INTENT(IN) :: coords(*)
14
          INTEGER, INTENT(OUT) :: rank
15
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
      Fortran binding
18
     MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
19
          INTEGER COMM, COORDS(*), RANK, IERROR
20
          For a process group with Cartesian structure, the function MPI_CART_RANK trans-
21
     lates the logical process coordinates to process ranks as they are used by the point-to-point
22
      routines.
23
          For dimension i with periods(i) = true, if the coordinate, coords(i), is out of range, that
24
      is, coords(i) < 0 or coords(i) \ge dims(i), it is shifted back to the interval
25
      0 \leq coords(i) < dims(i) automatically. Out-of-range coordinates are erroneous for non-
26
      periodic dimensions.
27
          If comm is associated with a zero-dimensional Cartesian topology, coords is not signif-
28
      icant and 0 is returned in rank.
29
30
^{31}
      MPI_CART_COORDS(comm, rank, maxdims, coords)
32
       IN
                                               communicator with Cartesian structure (handle)
33
                  comm
34
        IN
                  rank
                                               rank of a process within group of comm (integer)
35
       IN
                  maxdims
                                               length of vector coords in the calling program
36
                                               (integer)
37
       OUT
                                               integer array (of size maxdims) containing the
38
                  coords
                                               Cartesian coordinates of specified process (array of
39
40
                                               integers)
41
42
     C binding
43
      int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
44
      Fortran 2008 binding
45
      MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
46
          TYPE(MPI_Comm), INTENT(IN) :: comm
47
          INTEGER, INTENT(IN) :: rank, maxdims
48
```

	GER, INTENT(OUT) :: coord GER, OPTIONAL, INTENT(OUT		1 2
Transformer 1	- !!!		3
Fortran l	_COORDS(COMM, RANK, MAXDI		4
	GER COMM, RANK, MAXDIMS,		5
11111	ER COFFI, RANK, FIANDIFIS,	COORDS(*), TERROR	6
	· · · · · · · · · · · · · · · · · · ·	dinates translation is provided by	7
_	T_COORDS.		8 9
		-dimensional Cartesian topology,	9 10
coords will	be unchanged.		11
			12
MPI_GRA	PH_NEIGHBORS_COUNT(cor	nm, rank, nneighbors)	13
IN	comm	communicator with graph topology (handle)	14
IN	rank	rank of process in group of comm (integer)	15
OUT	nneighbors	number of neighbors of specified process (integer)	16 17
001	meighbols	number of neighbors of specified process (meeger)	18
C bindin	φ.		19
	0	_Comm comm, int rank, int *nneighbors)	20
			21
	2008 binding n_neighbors_count(comm, r	and preighborg iconver)	22
-	(MPI_Comm), INTENT(IN) ::	-	23
	GER, INTENT(IN) :: rank	Comm	24
INTEGER, INTENT(OUT) :: nneighbors			25
	GER, OPTIONAL, INTENT(OUT		26
			27 28
Fortran l	H_NEIGHBORS_COUNT(COMM, R	ANV NNETCUDADO TEDDAD)	29
	GER COMM, RANK, NNEIGHBOR		30
			31
			32
MPI GRA	PH_NEIGHBORS(comm, rank,	maxneighbors, neighbors)	33
- IN	comm	communicator with graph topology (handle)	34
			35
IN	rank	rank of process in group of comm (integer)	36
IN	maxneighbors	size of array neighbors (integer)	37 38
OUT	neighbors	ranks of processes that are neighbors to specified	39
		process (array of integers)	40
			41
C bindin	g		42
int MPI_(Graph_neighbors(MPI_Comm	comm, int rank, int maxneighbors,	43
	<pre>int neighbors[])</pre>		44
Fortran 2	2008 binding		45
	0	axneighbors, neighbors, ierror)	46
-	(MPI_Comm), INTENT(IN) ::	5	47
			48

1	INTEGER, INTENT(IN) :: rank, maxneighbors
2	INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
3	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4	
5	Fortran binding
6	MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
7	INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
8	
	MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency
9	information for a graph topology. The returned count and array of neighbors for the queried
10	rank will both include <i>all</i> neighbors and reflect the same edge ordering as was specified by
11	the original call to MPI_GRAPH_CREATE. Specifically, MPI_GRAPH_NEIGHBORS_COUNT
12	and MPI_GRAPH_NEIGHBORS will return values based on the original index and edges array
13	passed to MPI_GRAPH_CREATE (for the purpose of this example, we assume that index[-1]
14	is zero):
15	
16	• The number of neighbors (nneighbors) returned from
17	$MPI_GRAPH_NEIGHBORS_COUNT \text{ will } \mathrm{be} \ (index[rank] - index[rank-1]).$
18	
19	• The neighbors array returned from MPI_GRAPH_NEIGHBORS will be edges[index[rank-
20	1]] through edges[index[rank]-1].
21	
22	Example 8.5 Assume there are four processes 0, 1, 2, 3 with the following adjacency
23	matrix (note that some neighbors are listed multiple times):
24	
25	process neighbors
26	$0 \qquad 1, 1, 3$
27	1 $0, 0$
28	3 0, 2, 2
29	
30	Thus, the input arguments to MPI_GRAPH_CREATE are:
31	nnodes = 4
32	index = 3, 5, 6, 9
33	edges = 1, 1, 3, 0, 0, 3, 0, 2, 2
34	
35	Therefore, calling MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS for
36	each of the 4 processes will return:
37	Input rank Count Neighbors
38	
39	$egin{array}{cccccccccccccccccccccccccccccccccccc$
40	
41	$\begin{array}{cccccccccccccccccccccccccccccccccccc$
42	
43	
44	Example 8.6 Suppose that comm is a communicator with a shuffle-exchange topology.
45	The group has 2^n members. Each process is labeled by a_1, \ldots, a_n with $a_i \in \{0, 1\}$, and has
46	three neighbors: exchange $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \overline{a_n}$ $(\overline{a} = 1 - a)$, shuffle $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \overline{a_n}$
47	a_2, \ldots, a_n, a_1 , and unshuffle $(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$. The graph adjacency list is

Unofficial Draft for Comment Only

r	node	exchange	shuffle	unshuffle
		neighbors(1)	neighbors(2)	neighbors(3)
0	(000)	1	0	0
1	(001)	0	2	4
2	(010)	3	4	1
3	(011)	2	6	5
4	(100)	5	1	2
5	(101)	4	3	6
6	(110)	7	5	3
7	(111)	6	7	7

Suppose that the communicator comm has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

! assume: each process has stored a real number A.
! extract neighborhood information
CALL MPI_COMM_RANK(comm, myrank, ierr)
CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
! perform exchange permutation
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0, &
<pre>neighbors(1), 0, comm, status, ierr)</pre>
! perform shuffle permutation
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0, &
<pre>neighbors(3), 0, comm, status, ierr)</pre>
! perform unshuffle permutation
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0, &
<pre>neighbors(2), 0, comm, status, ierr)</pre>

```
MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS pro-
vide adjacency information for a distributed graph topology.
```

MPI_DIST	_GRAPH_NEIGHBORS_COUN	IT(comm, indegree, outdegree, weighted)	33
IN	comm	communicator with distributed graph topology	34
	comm	(handle)	35
		(numero)	36
OUT	indegree	number of edges into this process (non-negative	37
		integer)	38
OUT	outdegree	number of edges out of this process (non-negative	39
		integer)	40
OUT	weighted	false if MPI_UNWEIGHTED was supplied during	41
		creation, true otherwise (logical)	42
		()	43
			44
C binding	r		

int MDI Digt granh neighborg count (MDI Comm comm int windograp	
<pre>int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,</pre>	46
<pre>int *outdegree, int *weighted)</pre>	47

Fortran 2008 binding

Unofficial Draft for Comment Only

1 2 3 4 5 6	TYPE(INTEC LOGIC INTEC	(MPI_Comm), INTENT(IN) :: GER, INTENT(OUT) :: indeg CAL, INTENT(OUT) :: weigh GER, OPTIONAL, INTENT(OUT	ree, outdegree ted
7 8 9 10 11	INTEC	8	MM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR) GREE, IERROR
12 13 14	MPI_DIST	GRAPH_NEIGHBORS(comm	n, maxindegree, sources, sourceweights, ons, destweights)
15 16	IN	comm	communicator with distributed graph topology (handle)
17 18 19	IN	maxindegree	size of sources and sourceweights arrays (non-negative integer)
20 21	OUT	sources	processes for which the calling process is a destination (array of non-negative integers)
22 23	OUT	sourceweights	weights of the edges into the calling process (array of non-negative integers)
24 25 26	IN	maxoutdegree	size of destinations and destweights arrays (non-negative integer)
20 27 28	OUT	destinations	processes for which the calling process is a source (array of non-negative integers)
29 30	OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)
31 32 33 34 35	C bindin int MPI_I) Dist_graph_neighbors(MPI_	Comm comm, int maxindegree, int sources[], , int maxoutdegree, int destinations[],
36 37 38 39 40 41 42 43 44 45	MPI_Dist_ TYPE(INTEC INTEC INTEC	. .	legree, maxoutdegree es(maxindegree), degree) destweights(*)
46 47 48	Fortran k MPI_DIST_	_GRAPH_NEIGHBORS(COMM, MA	XINDEGREE, SOURCES, SOURCEWEIGHTS, NATIONS, DESTWEIGHTS, IERROR)

INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), IERROR

These calls are local. The number of edges into and out of the process returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT are the total number of such edges given in the call to MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE (potentially by processes other than the calling process in the case of MPI_DIST_GRAPH_CREATE). Multiply defined edges are all counted and returned by MPI_DIST_GRAPH_NEIGHBORS in some order. If MPI_UNWEIGHTED is supplied for sourceweights or destweights or both, or if MPI_UNWEIGHTED was supplied during the construction of the graph then no weight information is returned in that array or those arrays. If the communicator was created with MPI_DIST_GRAPH_CREATE_ADJACENT then for each rank in comm, the order of the values in sources and destinations is identical to the input that was used by the process with the same rank in **comm** old in the creation call. If the communicator was created with MPI_DIST_GRAPH_CREATE then the only requirement on the order of values in sources and destinations is that two calls to the routine with same input argument comm will return the same sequence of edges. If maxindegree or maxoutdegree is smaller than the numbers returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT, then only the first part of the full list is returned.

Advice to implementors. Since the query calls are defined to be local, each process needs to store the list of its neighbors with incoming and outgoing edges. Communication is required at the collective MPI_DIST_GRAPH_CREATE call in order to compute the neighbor lists for each process from the distributed graph specification. (End of advice to implementors.)

8.5.6 Cartesian Shift Coordinates

If the process topology is a Cartesian structure, an MPI_SENDRECV operation may be used along a coordinate direction to perform a shift of data. As input, MPI_SENDRECV takes the rank of a source process for the receive, and the rank of a destination process for the send. If the function MPI_CART_SHIFT is called for a Cartesian process group, it provides the calling process with the above identifiers, which then can be passed to MPI_SENDRECV. The user specifies the coordinate direction and the size of the step (positive or negative). The function is local.

MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)			
IN	comm	communicator with Cartesian structure (handle)	38
IN	direction	coordinate dimension of shift (integer)	39 40
IN	disp	displacement (> 0: upwards shift, < 0 : downwards shift) (integer)	$41 \\ 42$
OUT	rank_source	rank of source process (integer)	43
OUT	rank_dest	rank of destination process (integer)	44
001	Tank_dest	Taik of desimation process (integer)	$45 \\ 46$
C binding			
			47

1 $\mathbf{2}$

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19 20

21

22

23

24

2526

27 28

29

30

 31

32

33

34

35 36

```
1
     int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
\mathbf{2}
                     int *rank_source, int *rank_dest)
3
     Fortran 2008 binding
4
     MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
5
          TYPE(MPI_Comm), INTENT(IN) :: comm
6
          INTEGER, INTENT(IN) :: direction, disp
7
          INTEGER, INTENT(OUT) :: rank_source, rank_dest
8
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
     Fortran binding
11
     MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
12
          INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
13
          The direction argument indicates the coordinate dimension to be traversed by the shift.
14
     The dimensions are numbered from 0 to ndims-1, where ndims is the number of dimensions.
15
          Depending on the periodicity of the Cartesian group in the specified coordinate direc-
16
     tion, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift. In the case
17
     of an end-off shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest,
18
     indicating that the source or the destination for the shift is out of range.
19
          It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or
20
     greater than or equal to the number of dimensions in the Cartesian communicator. This
21
     implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with
22
     a zero-dimensional Cartesian topology.
23
^{24}
     Example 8.7 The communicator, comm, has a two-dimensional, periodic, Cartesian topol-
25
     ogy associated with it. A two-dimensional array of REALs is stored one element per process,
26
     in variable A. One wishes to skew this array, by shifting column i (vertically, i.e., along the
27
     column) by i steps.
28
29
30
     ! find process rank
^{31}
     CALL MPI_COMM_RANK(comm, rank, ierr)
32
     ! find Cartesian coordinates
33
     CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
34
     ! compute shift source and destination
35
     CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
36
     ! skew array
37
     CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm, &
38
                                   status, ierr)
39
40
           Advice to users. In Fortran, the dimension indicated by DIRECTION = i has DIMS(i+1)
41
           nodes, where DIMS is the array that was used to create the grid. In C, the dimension
42
           indicated by direction = i is the dimension specified by dims[i]. (End of advice
43
           to users.)
44
45
46
47
48
```

8.5.7 Partitioning of Cartesian Structures

MPI_CART_SUB(comm,	remain_dims,	newcomm)	
--------------------	--------------	----------	--

IN	comm	communicator with Cartesian structure (handle)
IN	remain_dims	the i-th entry of remain_dims specifies whether the
		i-th dimension is kept in the subgrid $(true)$ or is
		dropped (false) (array of logicals)
OUT	newcomm	communicator containing the subgrid that includes
		the calling process (handle)

C binding

int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)

Fortran 2008 binding

MPI_Cart_sub(comm, remain_dims, newcomm, ierror)	
TYPE(MPI_Comm), INTENT(IN) :: comm	
LOGICAL, INTENT(IN) :: remain_dims(*)	
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	

Fortran binding

MPI_	_CART_SUB	B(COMM,	REMAIN_I	DIMS,	NEWCOMM,	IERROR)
	INTEGER	COMM,	NEWCOMM,	IERRO)r	
	LOGICAL	REMAIN	I_DIMS(*)			

If a Cartesian topology has been created with MPI_CART_CREATE, the function MPI_CART_SUB can be used to partition the communicator group into subgroups that form lower-dimensional Cartesian subgrids, and to build for each subgroup a communicator with the associated subgrid Cartesian topology. If all entries in remain_dims are false or comm is already associated with a zero-dimensional Cartesian topology. (This function is closely related to MPI_COMM_SPLIT.)

Example 8.8 Assume that MPI_Cart_create(..., comm) has defined a $(2 \times 3 \times 4)$ grid. Let remain_dims = (true, false, true). Then a call to

MPI_Cart_sub(comm, remain_dims, newcomm)

will create three communicators each with eight processes in a 2×4 Cartesian topology. If remain_dims = (false, false, true) then the call to

MPI_Cart_sub(comm, remain_dims, newcomm)

will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.

Unofficial Draft for Comment Only

 $41 \\ 42$

```
1
     8.5.8
             Low-Level Topology Functions
\mathbf{2}
      The two additional functions introduced in this section can be used to implement all other
3
      topology functions. In general they will not be called by the user directly, unless he or she
4
      is creating additional virtual topology capability other than that provided by MPI. The two
5
      calls are both local.
6
7
8
      MPI_CART_MAP(comm, ndims, dims, periods, newrank)
9
       IN
                 comm
                                              input communicator (handle)
10
       IN
                 ndims
                                              number of dimensions of Cartesian structure (integer)
11
12
       IN
                 dims
                                              integer array of size ndims specifying the number of
13
                                               processes in each coordinate direction
14
       IN
                  periods
                                              logical array of size ndims specifying the periodicity
15
                                               specification in each coordinate direction
16
       OUT
                 newrank
                                               reordered rank of the calling process;
17
                                               MPI_UNDEFINED if calling process does not belong
18
19
                                               to grid (integer)
20
21
      C binding
22
      int MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],
23
                     const int periods[], int *newrank)
24
      Fortran 2008 binding
25
     MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)
26
          TYPE(MPI_Comm), INTENT(IN) :: comm
27
          INTEGER, INTENT(IN) :: ndims, dims(ndims)
28
          LOGICAL, INTENT(IN) :: periods(ndims)
29
          INTEGER, INTENT(OUT) :: newrank
30
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
      Fortran binding
33
     MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
34
          INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
35
          LOGICAL PERIODS(*)
36
          MPI_CART_MAP computes an "optimal" placement for the calling process on the phys-
37
      ical machine. A possible implementation of this function is to always return the rank of the
38
      calling process, that is, not to perform any reordering.
39
40
           Advice to implementors.
                                       The function MPI_CART_CREATE(comm, ndims, dims,
41
           periods, reorder, comm_cart), with reorder = true can be implemented by calling
42
           MPI_CART_MAP(comm, ndims, dims, periods, newrank), then calling
43
           MPI_COMM_SPLIT(comm, color, key, comm_cart), with color = 0 if newrank \neq
44
           MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank. If ndims
45
           is zero-then a zero-dimensional Cartesian topology is created.
46
47
           The function MPI_CART_SUB(comm, remain_dims, comm_new) can be implemented
           by a call to MPI_COMM_SPLIT(comm, color, key, comm_new), using a single number
48
```

1 encoding of the lost dimensions as color and a single number encoding of the preserved $\mathbf{2}$ dimensions as key. 3 All other Cartesian topology functions can be implemented locally, using the topology 4 information that is cached with the communicator. (End of advice to implementors.) 56 The corresponding function for graph structures is as follows. 7 8 MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank) 9 10 IN comm input communicator (handle) 11 IN nnodes number of graph nodes (integer) 12integer array specifying the graph structure, see IN index 13 MPI_GRAPH_CREATE 1415IN edges integer array specifying the graph structure 16reordered rank of the calling process; OUT newrank 17 MPI_UNDEFINED if the calling process does not 18 belong to graph (integer) 19 20C binding 21int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[], 22 const int edges[], int *newrank) 23 24 Fortran 2008 binding 25MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror) 26TYPE(MPI_Comm), INTENT(IN) :: comm 27INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*) 28 INTEGER, INTENT(OUT) :: newrank 29INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 Fortran binding 31MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) 32 INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR 33 34 35Advice to implementors. The function MPI_GRAPH_CREATE(comm, nnodes, index, 36 edges, reorder, comm_graph), with reorder = true can be implemented by calling 37 MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank), then calling 38 MPI_COMM_SPLIT(comm, color, key, comm_graph), with color = 0 if newrank \neq 39 MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank. 40 All other graph topology functions can be implemented locally, using the topology 41 information that is cached with the communicator. (End of advice to implementors.) 4243 8.6 Neighborhood Collective Communication on Process Topologies 4445

MPI process topologies specify a communication graph, but they implement no communication function themselves. Many applications require sparse nearest neighbor communications that can be expressed as graph topologies. We now describe several collective 48

Unofficial Draft for Comment Only

1 operations that perform communication along the edges of a process topology. All of these $\mathbf{2}$ functions are collective; i.e., they must be called by all processes in the specified commu-3 nicator. See Section 6 for an overview of other dense (global) collective communication 4 operations and the semantics of collective operations.

5If the graph was created with MPI_DIST_GRAPH_CREATE_ADJACENT with sources 6 and destinations containing $0, \ldots, n-1$, where n is the number of processes in the group $\overline{7}$ of comm_old (i.e., the graph is fully connected and also includes an edge from each node 8 to itself), then the sparse neighborhood communication routine performs the same data 9 exchange as the corresponding dense (fully-connected) collective operation. In the case of a 10 Cartesian communicator, only nearest neighbor communication is provided, corresponding 11to rank_source and rank_dest in MPI_CART_SHIFT with input disp = 1.

Neighborhood collective communications enable communication on a

process topology. This high-level specification of data exchange among neighboring

processes enables optimizations in the MPI library because the communication pattern

is known statically (the topology). Thus, the implementation can compute optimized

message schedules during creation of the topology [39]. This functionality can signif-

icantly simplify the implementation of neighbor exchanges [35]. (End of rationale.)

12

Rationale.

13 14

1516

17

18

19

For a distributed graph topology, created with MPI_DIST_GRAPH_CREATE, the se-20quence of neighbors in the send and receive buffers at each process is defined as the sequence 21returned by MPI_DIST_GRAPH_NEIGHBORS for destinations and sources, respectively. For 22 a general graph topology, created with MPI_GRAPH_CREATE, the use of neighborhood col-23lective communication is restricted to adjacency matrices, where the number of edges be- 24 tween any two processes is defined to be the same for both processes (i.e., with a symmetric 25adjacency matrix). In this case, the order of neighbors in the send and receive buffers is 26defined as the sequence of neighbors as returned by MPI_GRAPH_NEIGHBORS. Note that 27general graph topologies should generally be replaced by the distributed graph topologies. 28

For a Cartesian topology, created with MPI_CART_CREATE, the sequence of neigh-29 bors in the send and receive buffers at each process is defined by order of the dimensions, 30 first the neighbor in the negative direction and then in the positive direction with dis- 31 placement 1. The numbers of sources and destinations in the communication routines are 32 2*ndims with ndims defined in MPI_CART_CREATE. If a neighbor does not exist, i.e., at 33 the border of a Cartesian topology in the case of a non-periodic virtual grid dimension (i.e., 34 periods[...]==false), then this neighbor is defined to be MPI_PROC_NULL. 35

If a neighbor in any of the functions is MPI_PROC_NULL, then the neighborhood collec-36 tive communication behaves like a point-to-point communication with MPI_PROC_NULL in 37 this direction. That is, the buffer is still part of the sequence of neighbors but it is neither 38 communicated nor updated. 39

40 41

Neighborhood Gather 8.6.1

42In this function, each process i gathers data items from each process j if an edge (j, i) exists 43 in the topology graph, and each process i sends the same data items to all processes j where 44 an edge (i, j) exists. The send buffer is sent to each neighboring process and the *l*-th block 45in the receive buffer is received from the *l*-th neighbor.

- 46
- 4748

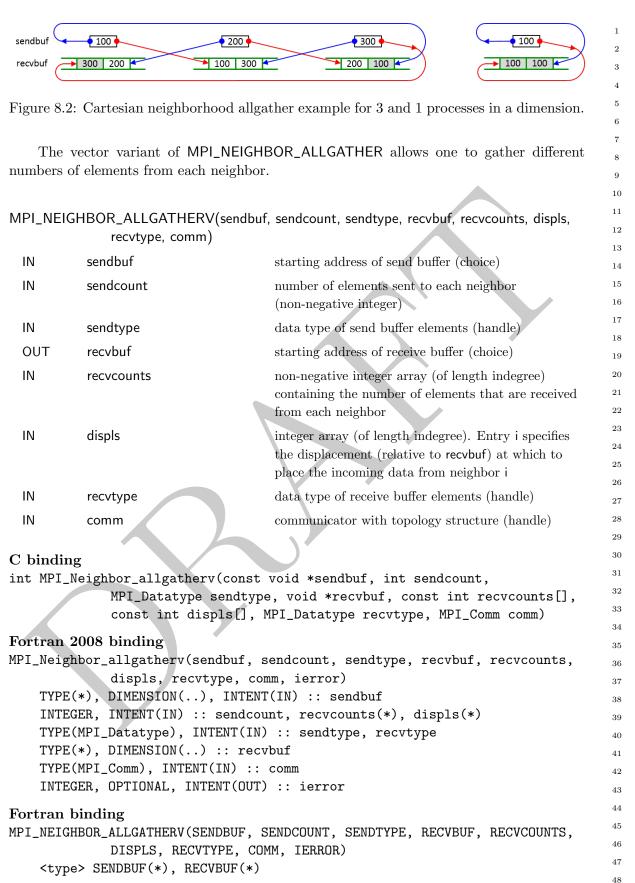
MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 1 2 comm) 3 sendbuf IN starting address of send buffer (choice) 4 IN sendcount number of elements sent to each neighbor 5 (non-negative integer) 6 IN sendtype data type of send buffer elements (handle) OUT recvbuf starting address of receive buffer (choice) number of elements received from each neighbor IN recvcount 10 (non-negative integer) 11 IN recvtype data type of receive buffer elements (handle) 1213 IN comm communicator with topology structure (handle) 1415C binding 16int MPI_Neighbor_allgather(const void *sendbuf, int sendcount, 17 MPI_Datatype sendtype, void *recvbuf, int recvcount, 18 MPI_Datatype recvtype, MPI_Comm comm) 19 Fortran 2008 binding 2021MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, 22 recvtype, comm, ierror) 23TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf 24 INTEGER, INTENT(IN) :: sendcount, recvcount 25TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 26TYPE(*), DIMENSION(..) :: recvbuf 27TYPE(MPI_Comm), INTENT(IN) :: comm 28 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29Fortran binding 30 MPI_NEIGHBOR_ALLGATHER (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 31RECVTYPE, COMM, IERROR) 32 <type> SENDBUF(*), RECVBUF(*) 33 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 34 35This function supports Cartesian communicators, graph communicators, and distributed 36 graph communicators as described in Section 8.6. If comm is a distributed graph commu-37 nicator, the outcome is as if each process executed sends to each of its outgoing neighbors 38 and receives from each of its incoming neighbors: 39 40 MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted); 41 int *srcs=(int*)malloc(indegree*sizeof(int)); 42int *dsts=(int*)malloc(outdegree*sizeof(int)); MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED, 43 44 outdegree, dsts, MPI_UNWEIGHTED); 45int k,l; 4647/* assume sendbuf and recvbuf are of type (char*) */

Unofficial Draft for Comment Only

for(k=0; k<outdegree; ++k)</pre>

1	MDT Taand (aandhuf aandaaunt aandtuna data[4]).
2	<pre>MPI_Isend(sendbuf, sendcount, sendtype,dsts[k],);</pre>
3	<pre>for(1=0; 1<indegree; ++1)<="" pre=""></indegree;></pre>
4	<pre>MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,</pre>
5	srcs[1],);
6	
7	MPI_Waitall();
8	Figure 8.1 above the neighborhood gether communication of one proceed with outgoing
9	Figure 8.1 shows the neighborhood gather communication of one process with outgoing neighbors $d_0 \ldots d_3$ and incoming neighbors $s_0 \ldots s_5$. The process will send its sendbuf to
10	all four destinations (outgoing neighbors) and it will receive the contribution from all six
11	sources (incoming neighbors) into separate locations of its receive buffer.
12	sources (meetining heighbors) mus separate rocations of his receive buildr.
13	d_0
14	d_2, s_4
15	s_0
16 17	
18	$d_1 \longleftarrow s_1$
19	
20	
21	
22	d_3, s_5
23	sendbuf
24	
25	
26	s_0 s_1 s_2 s_3 s_4 s_5
27	recvbuf
28 29	
30	Figure 8.1: Neighborhood gather communication example.
31	
32	All arguments are significant on all processes and the argument comm must have iden- tical values on all processes.
33	The type signature associated with sendcount, sendtype, at a process must be equal to
34	the type signature associated with schedount, schedope, at a process must be equal to the type signature associated with recvcount, recvtype at all other processes. This implies
35	that the amount of data sent must be equal to the amount of data received, pairwise between
36	every pair of communicating processes. Distinct type maps between sender and receiver are
37	still allowed.
38	
39	Rationale. For optimization reasons, the same type signature is required indepen-
40	dently of whether the topology graph is connected or not. (<i>End of rationale.</i>)
41 42	The "in place" option is not meaningful for this operation.
42	Encourse a Que of Contaction with the buffer was a since direction of with
44	Example 8.9 On a Cartesian virtual grid, the buffer usage in a given direction d with dims[d]==3 and 1, respectively during creation of the communicator is described in Fig-
45	$\dim[q] = 5$ and 1, respectively during creation of the communicator is described in Figure 8.2.
46	The figure may apply to any (or multiple) directions in the Cartesian topology. The grey
47	buffers are required in all cases but are only accessed if during creation of the communicator,
48	periods[d] was defined as 1 (in C) or .TRUE. (in Fortran).

Unofficial Draft for Comment Only



1	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,			
2	IERROR			
3				
4	This function supports Cartesian communicators, graph communicators, and distributed			
5	graph communicators as described in Section 8.6. If comm is a distributed graph commu-			
6	nicator, the outcome is as if each process executed sends to each of its outgoing neighbors			
7	and receives from each of its incoming neighbors:			
8	MDT Digt graph noighborg count(comp kindograph koutdograph kucighted).			
9	<pre>MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted); int *srcs=(int*)malloc(indegree*sizeof(int));</pre>			
10	Ĵ			
11	<pre>int *dsts=(int*)malloc(outdegree*sizeof(int));</pre>			
12	<pre>MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,</pre>			
13	<pre>outdegree, dsts, MPI_UNWEIGHTED);</pre>			
14	int k,l;			
15				
16	<pre>/* assume sendbuf and recvbuf are of type (char*) */</pre>			
17	<pre>for(k=0; k<outdegree; ++k)<="" pre=""></outdegree;></pre>			
18	<pre>MPI_Isend(sendbuf, sendcount, sendtype, dsts[k],);</pre>			
19				
20	<pre>for(1=0; 1<indegree; ++1)<="" pre=""></indegree;></pre>			
21	<pre>MPI_Irecv(recvbuf+displs[l]*extent(recvtype), recvcounts[l], recvtype,</pre>			
22	<pre>srcs[1],);</pre>			
23				
24	<pre>MPI_Waitall();</pre>			

The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[l], recvtype at any other process with srcs[l]==j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed. The data received from the l-th neighbor is placed into recvbuf beginning at offset displs[l] elements (in terms of the recvtype).

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument **comm** must have identical values on all processes.

³⁵₃₆ 8.6.2 Neighbor Alltoall

³⁷ In this function, each process i receives data items from each process j if an edge (j,i)³⁸ exists in the topology graph or Cartesian topology. Similarly, each process i sends data ³⁹ items to all processes j where an edge (i, j) exists. This call is more general than

⁴⁰ MPI_NEIGHBOR_ALLGATHER in that different data items can be sent to each neighbor. ⁴¹ The *k*-th block in send buffer is sent to the *k*-th neighboring process and the *l*-th block in ⁴² the receive buffer is received from the *l*-th neighbor.

43 44

25

26

27

28

29

30

 31

32

33

- 45
- 46
- 47
- 48

1 MPI_NEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 2 comm) sendbuf IN starting address of send buffer (choice) IN sendcount number of elements sent to each neighbor 5 (non-negative integer) 6 IN sendtype data type of send buffer elements (handle) OUT recvbuf starting address of receive buffer (choice) number of elements received from each neighbor IN recvcount 10 (non-negative integer) 11 IN recvtype data type of receive buffer elements (handle) 1213 IN comm communicator with topology structure (handle) 1415C binding 16int MPI_Neighbor_alltoall(const void *sendbuf, int sendcount, 17 MPI_Datatype sendtype, void *recvbuf, int recvcount, 18 MPI_Datatype recvtype, MPI_Comm comm) 19 20Fortran 2008 binding 21MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, 22 recvtype, comm, ierror) 23TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf 24 INTEGER, INTENT(IN) :: sendcount, recvcount 25TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 26TYPE(*), DIMENSION(..) :: recvbuf 27TYPE(MPI_Comm), INTENT(IN) :: comm 28 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29Fortran binding 30 MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 31RECVTYPE, COMM, IERROR) 32 <type> SENDBUF(*), RECVBUF(*) 33 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 34 35This function supports Cartesian communicators, graph communicators, and distributed 36 graph communicators as described in Section 8.6. If comm is a distributed graph commu-37 nicator, the outcome is as if each process executed sends to each of its outgoing neighbors 38 and receives from each of its incoming neighbors: 39 40 MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted); 41 int *srcs=(int*)malloc(indegree*sizeof(int)); 42int *dsts=(int*)malloc(outdegree*sizeof(int)); MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED, 43 44 outdegree, dsts, MPI_UNWEIGHTED); 45int k,l; 4647/* assume sendbuf and recvbuf are of type (char*) */

Unofficial Draft for Comment Only

for(k=0; k<outdegree; ++k)</pre>

```
for(l=0; l<indegree; ++1)</pre>
```

MPI_Waitall(...);

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument **comm** must have identical values on all processes.

Example 8.10 For a halo communication on a Cartesian grid, the buffer usage in a given direction d with dims[d]==3 and 1, respectively during creation of the communicator is described in Figure 8.3.

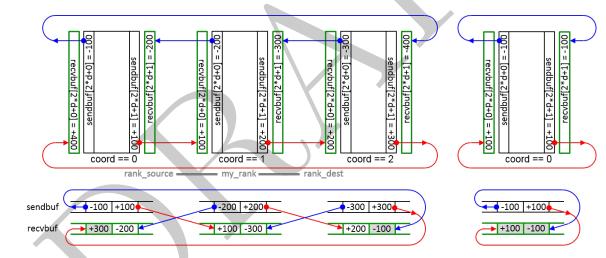


Figure 8.3: Cartesian neighborhood alltoall example for 3 and 1 processes in a dimension.

The figure may apply to any (or multiple) directions in the Cartesian topology. The grey buffers are required in all cases but are only accessed if during creation of the communicator, periods[d] was defined as 1 (in C) or .TRUE. (in Fortran).

If each array element of sendbuf and recvbuf are described by sendcount, sendtype and recvbuf, recvtype, then after MPI_NEIGHBOR_ALLTOALL on a Cartesian communicator returned, the content of the recvbuf is as if the following code is executed:

```
44
45 MPI_Cartdim_get(comm, &ndims);
```

```
_{46} for( /*direction*/ d=0; d < ndims; d++) {
```

```
MPI_Cart_shift(comm, /*direction*/ d, /*disp*/ 1, &rank_source, &rank_dest);
```

```
MPI_Sendrecv(sendbuf[d*2+0],sendcount,sendtype,<u>rank_source</u>,/*sendtag*/d*2,
```

 $\mathbf{2}$

```
1
             recvbuf[d*2+1],recvcount,recvtype,rank_dest, /*recvtag*/ d*2,
                                                                                    \mathbf{2}
              comm,&status);/*communication in direction of displacment -1*/
                                                                                    3
MPI_Sendrecv(sendbuf[d*2+1],sendcount,sendtype,rank_dest, /*sendtag*/ d*2+1,
                                                                                    4
             recvbuf[d*2+0],recvcount,recvtype,rank_source,/*recvtag*/d*2+1,
                                                                                    \mathbf{5}
             comm,&status);/*communication in direction of displacment +1*/
                                                                                    6
```

```
}
```

The first call to MPI_Sendrecv implements the upper (blue) communication pattern in each diagram of Figure 8.3, whereas the second call is for the lower (red) pattern.

Advice to implementors. For a Cartesian topology, if the virtual grid in a direction d is periodic and dims[d] is equal to 0 or 1, then rank_source and rank_dest are identical, but still all ndims send and ndims receive operations use different buffers. If in this case, the two send and receive operations per direction or of all directions are internally parallelized, then the several send and receive operations for the same sender-receiver process pair must be initiated in the same sequence on sender and receiver side or they shall be distinguished by different tags. The code above shows a valid sequence of operations and tags. (End of advice to implementors.)

The vector variant of MPI_NEIGHBOR_ALLTOALL allows sending/receiving different numbers of elements to and from each neighbor.

MPI_NEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)			
IN	sendbuf	starting address of send buffer (choice)	
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each	
IN	sdispls	neighbor integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which to send the outgoing data to neighbor j	
IN	sendtype	data type of send buffer elements (handle)	
Ουτ	recvbuf	starting address of receive buffer (choice)	
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received	

from each neighbor

integer array (of length indegree). Entry i specifies

the displacement (relative to recvbuf) at which to

place the incoming data from neighbor i

data type of receive buffer elements (handle)

communicator with topology structure (handle)

IN

IN

IN

rdispls

recvtype

comm

int MPI_Neighbor_alltoallv(const void *sendbuf, const int sendcounts[], const int sdispls[], MPI_Datatype sendtype, void *recvbuf, $\overline{7}$

8 9

10

11

12

13

14

15

16

17 18

19

38

39

40

41

42

47

```
1
                    const int recvcounts[], const int rdispls[],
\mathbf{2}
                    MPI_Datatype recvtype, MPI_Comm comm)
3
     Fortran 2008 binding
4
     MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
5
                    recvcounts, rdispls, recvtype, comm, ierror)
6
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
7
          INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
8
                    rdispls(*)
9
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
10
         TYPE(*), DIMENSION(...) :: recvbuf
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
16
                    RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
17
          <type> SENDBUF(*), RECVBUF(*)
18
          INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
19
                     RECVTYPE, COMM, IERROR
20
         This function supports Cartesian communicators, graph communicators, and distributed
21
     graph communicators as described in Section 8.6. If comm is a distributed graph commu-
22
     nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
23
     and receives from each of its incoming neighbors:
24
25
     MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
26
     int *srcs=(int*)malloc(indegree*sizeof(int));
27
     int *dsts=(int*)malloc(outdegree*sizeof(int));
28
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
29
                                 outdegree, dsts, MPI_UNWEIGHTED);
30
     int k,l;
^{31}
32
     /* assume sendbuf and recvbuf are of type (char*) */
33
     for(k=0; k<outdegree; ++k)</pre>
34
       MPI_Isend(sendbuf+sdispls[k]*extent(sendtype), sendcounts[k], sendtype,
35
                  dsts[k],...);
36
37
     for(l=0; l<indegree; ++1)</pre>
38
       MPI_Irecv(recvbuf+rdispls[1]*extent(recvtype), recvcounts[1], recvtype,
39
                  srcs[1],...);
40
41
     MPI_Waitall(...);
42
43
         The type signature associated with sendcounts[k], sendtype with dsts[k] = = j at process
44
     i must be equal to the type signature associated with recvcounts[I], recvtype with srcs[I] = = i
45
     at process j. This implies that the amount of data sent must be equal to the amount of
```

data received, pairwise between every pair of communicating processes. Distinct type maps
 between sender and receiver are still allowed. The data in the sendbuf beginning at offset
 sdispls[k] elements (in terms of the sendtype) is sent to the k-th outgoing neighbor. The data

Unofficial Draft for Comment Only

received :	from the I-th incoming	; neighbor is placed into recvbuf beginning at offset rdispls[l]	1
elements	(in terms of the recvty	/pe).	2
The	"in place" option is no	ot meaningful for this operation.	3
All a	arguments are significar	nt on all processes and the argument comm must have iden-	4
tical valu	ies on all processes.		5
MPL	_NEIGHBOR_ALLTOA	LLW allows one to send and receive with different datatypes	6
to and fr	om each neighbor.		7
			8
MPI_NEI		(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,	9 10
	rdispls, recvtype	s, comm)	11
IN	sendbuf	starting address of send buffer (choice)	12
IN	sendcounts	non-negative integer array (of length outdegree)	13
		specifying the number of elements to send to each	14
		neighbor	15
IN	sdispls	integer array (of length outdegree). Entry j specifies	16
11 N	3013013	the displacement in bytes (relative to sendbuf) from	17 18
		which to take the outgoing data destined for	19
		neighbor j (array of integers)	20
IN	sendtypes	array of datatypes (of length outdegree). Entry j	20
IIN	senutypes	specifies the type of data to send to neighbor j (array	22
		of handles)	23
0.U.T			24
OUT	recvbuf	starting address of receive buffer (choice)	25
IN	recvcounts	non-negative integer array (of length indegree)	26
		specifying the number of elements that are received	27
		from each neighbor	28
IN	rdispls	integer array (of length indegree). Entry i specifies	29
		the displacement in bytes (relative to recvbuf) at	30
		which to place the incoming data from neighbor i	31
		(array of integers)	32
IN	recvtypes	array of datatypes (of length indegree). Entry i	33 34
		specifies the type of data received from neighbor i	34 35
		(array of handles)	36
IN	comm	communicator with topology structure (handle)	37
			38
C bindi	ng		39
	e e e e e e e e e e e e e e e e e e e	(const void *sendbuf, const int sendcounts[],	40
	const MPI_Ain	<pre>nt sdispls[], const MPI_Datatype sendtypes[],</pre>	41
	void *recvbuf	<pre>const int recvcounts[],</pre>	42
		<pre>nt rdispls[], const MPI_Datatype recvtypes[],</pre>	43
	MPI_Comm comm	1)	44
Fortran	2008 binding		45
		dbuf, sendcounts, sdispls, sendtypes, recvbuf,	46
	-	displs, recvtypes, comm, ierror)	47 48

```
1
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
\mathbf{2}
          INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
3
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
4
          TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
5
          TYPE(*), DIMENSION(..) :: recvbuf
6
          TYPE(MPI_Comm), INTENT(IN) :: comm
7
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     Fortran binding
9
     MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
10
                    RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)
11
          <type> SENDBUF(*), RECVBUF(*)
12
          INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
13
                     IERROR
14
          INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
15
16
         This function supports Cartesian communicators, graph communicators, and distributed
17
     graph communicators as described in Section 8.6. If comm is a distributed graph commu-
18
     nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
19
     and receives from each of its incoming neighbors:
20
21
     MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
22
     int *srcs=(int*)malloc(indegree*sizeof(int));
23
     int *dsts=(int*)malloc(outdegree*sizeof(int));
^{24}
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                                 outdegree, dsts, MPI_UNWEIGHTED);
25
26
     int k,l;
27
     /* assume sendbuf and recvbuf are of type (char*) */
28
     for(k=0; k<outdegree; ++k)</pre>
29
30
       MPI_Isend(sendbuf+sdispls[k], sendcounts[k], sendtypes[k], dsts[k],...);
^{31}
     for(1=0; 1<indegree; ++1)</pre>
32
33
       MPI_Irecv(recvbuf+rdispls[1], recvcounts[1], recvtypes[1], srcs[1],...);
34
     MPI_Waitall(...);
35
36
         The type signature associated with sendcounts[k], sendtypes[k] with dsts[k]==j at pro-
37
     cess i must be equal to the type signature associated with recvcounts[1], recvtypes[1] with
38
     srcs[I] == i at process j. This implies that the amount of data sent must be equal to the
39
     amount of data received, pairwise between every pair of communicating processes. Distinct
40
     type maps between sender and receiver are still allowed.
41
          The "in place" option is not meaningful for this operation.
42
         All arguments are significant on all processes and the argument comm must have iden-
43
     tical values on all processes.
44
45
46
47
48
```

8.7 Nonblocking Neighborhood Communication on Process Topologies

Nonblocking variants of the neighborhood collective operations allow relaxed synchronization and overlapping of computation and communication. The semantics are similar to nonblocking collective operations as described in Section 6.12.

8.7.1 Nonblocking Neighborhood Gather

MPI_INEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)

	, , , , ,		12
IN	sendbuf	starting address of send buffer (choice)	13
IN	sendcount	number of elements sent to each neighbor	14
		(non-negative integer)	15
IN	sendtype	data type of send buffer elements (handle)	16
OUT	recvbuf	starting address of reasing huffer (shoise)	17
001	recybui	starting address of receive buffer (choice)	18
IN	recvcount	number of elements received from each neighbor	19
		(non-negative integer)	20
IN	recvtype	data type of receive buffer elements (handle)	21
INI	comm	communication with top along structure (handle)	22
IN	comm	communicator with topology structure (handle)	23
OUT	request	communication request (handle)	24

C binding

int MPI_Ineighbor_allgather(const void *sendbuf, int sendcount, 2728 MPI_Datatype sendtype, void *recvbuf, int recvcount, 29MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 30 Fortran 2008 binding 31MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, 32 recvtype, comm, request, ierror) 33 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 34 INTEGER, INTENT(IN) :: sendcount, recvcount 35 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 36 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 37 TYPE(MPI_Comm), INTENT(IN) :: comm 38 TYPE(MPI_Request), INTENT(OUT) :: request 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 41 Fortran binding 42MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 43 RECVTYPE, COMM, REQUEST, IERROR) 44<type> SENDBUF(*), RECVBUF(*) 45INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 46This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHER. 47

1 2

3

4

5 6

9 10

11

25 26

1 2	MPI_INEIGHBOR_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request)			
3	IN	sendbuf	starting address of send buffer (choice)	
4 5 6	IN	sendcount	number of elements sent to each neighbor (non-negative integer)	
7	IN	sendtype	data type of send buffer elements (handle)	
8	OUT	recvbuf	starting address of receive buffer (choice)	
9 10 11 12	IN	recvcounts	non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor	
13 14 15	IN	displs	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i	
16 17	IN	recvtype	data type of receive buffer elements (handle)	
18	IN	comm	communicator with topology structure (handle)	
19	OUT	request	communication request (handle)	
C binding Int MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) Fortran 2008 binding MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN) :: sendcount TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvtype TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN), ASYNCHRONOUS :: recvbuf				
35	TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request			
36 37	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
38	Fortran binding			
39	MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,			
40 41	DISPLS, RECVTYPE, COMM, REQUEST, IERROR)			
42	<pre><type> SENDBUF(*), RECVBUF(*) </type></pre>			
43	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, REQUEST, IERROR			
44				
45	This c	all starts a nonblocking varia	nt of MPI_NEIGHBOR_ALLGATHERV.	
46 47				
48				

8.7. NONBLOCKING NEIGHBORHOOD COMMUNICATION 393 8.7.2 Nonblocking Neighborhood Alltoall 1 2 MPI_INEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 5 comm, request) 6 IN sendbuf starting address of send buffer (choice) 7 IN sendcount number of elements sent to each neighbor (non-negative integer) 9 10 sendtype data type of send buffer elements (handle) IN 11 OUT recvbuf starting address of receive buffer (choice) 12IN number of elements received from each neighbor recvcount 13 (non-negative integer) 1415IN recvtype data type of receive buffer elements (handle) 16communicator with topology structure (handle) IN comm 17OUT communication request (handle) request 18 19 C binding 2021int MPI_Ineighbor_alltoall(const void *sendbuf, int sendcount, 22 MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 23 24 Fortran 2008 binding 25MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, 26recvtype, comm, request, ierror) 27TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 28INTEGER, INTENT(IN) :: sendcount, recvcount 29 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 30 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 31TYPE(MPI_Comm), INTENT(IN) :: comm 32 TYPE(MPI_Request), INTENT(OUT) :: request 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 35 Fortran binding 36 MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 37 RECVTYPE, COMM, REQUEST, IERROR) 38 <type> SENDBUF(*), RECVBUF(*) 39 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 40 This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALL. 41 4243 44 4546

12	MPI_INEI	GHBOR_ALLTOALLV(sendburdispls, recvtype, comn	uf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, n, request)		
$\frac{3}{4}$	IN	sendbuf	starting address of send buffer (choice)		
5 6 7	IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor		
8 9 10	IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j		
11 12	IN	sendtype	data type of send buffer elements (handle)		
12	OUT	recvbuf	starting address of receive buffer (choice)		
14 15 16	IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor		
17 18 19 20	IN	rdispls	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i		
21	IN	recvtype	data type of receive buffer elements (handle)		
22	IN	comm	communicator with topology structure (handle)		
23 24	OUT	request	communication request (handle)		
26 27 28 29 30	C bindin int MPI_:	Ineighbor_alltoallv(cons const int sdispls[const int recvcoun	st void *sendbuf, const int sendcounts[],], MPI_Datatype sendtype, void *recvbuf, ts[], const int rdispls[], ype, MPI_Comm comm, MPI_Request *request)		
31	Fortran 2	2008 binding			
32	MPI_Inei		, sendcounts, sdispls, sendtype, recvbuf,		
33 34	TYDE		s, recvtype, comm, request, ierror) ENT(IN), ASYNCHRONOUS :: sendbuf		
35			RONOUS :: sendcounts(*), sdispls(*),		
36		recvcounts(*), rd	-		
37			IN) :: sendtype, recvtype		
38 39		(*), DIMENSION(), ASYN			
40	TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request				
41		GER, OPTIONAL, INTENT(OU	-		
42					
43	Fortran I		, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,		
44 45			S, RECVTYPE, COMM, REQUEST, IERROR)		
46	• -	e> SENDBUF(*), RECVBUF(>	*)		
47	INTEG		PLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),		
48		RECVTYPE, COMM, RE	EQUEST, IERROR		

This call starts a nonblocking variant of $\mathsf{MPI_NEIGHBOR_ALLTOALLV}.$

³ MPI_INEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, ⁴			
	rdispls, recvtypes, comm,		5
IN	sendbuf	starting address of send buffer (choice)	6
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor	7 8 9 10
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers)	10 11 12 13 14
IN	sendtypes	array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles)	15 16 17
OUT	recvbuf	starting address of receive buffer (choice)	18 19
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor	20 21 22
IN	rdispls	integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers)	23 24 25 26
IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles)	27 28 29 30
IN	comm	communicator with topology structure (handle)	31
OUT	request	communication request (handle)	32 33 34
<pre>C binding int MPI_Ineighbor_alltoallw(const void *sendbuf, const int sendcounts[],</pre>			
MPI_Ineig TYPE(INTEG	008 binding hbor_alltoallw(sendbuf, s recvcounts, rdispls, *), DIMENSION(), INTENT ER, INTENT(IN), ASYNCHRON	<pre>sendcounts, sdispls, sendtypes, recvbuf, recvtypes, comm, request, ierror) C(IN), ASYNCHRONOUS :: sendbuf NOUS :: sendcounts(*), recvcounts(*) , INTENT(IN), ASYNCHRONOUS :: sdispls(*),</pre>	40 41 42 43 44 45 46 47 48

Unofficial Draft for Comment Only

TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), 2 recvtypes(*) 3 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 4 TYPE(MPI_Comm), INTENT(IN) :: comm 5TYPE(MPI_Request), INTENT(OUT) :: request 6 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 7 Fortran binding 8 MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 9 RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR) 10 <type> SENDBUF(*), RECVBUF(*) 11 INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM, 12REQUEST, IERROR 13 INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*) 14 15This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLW. 1617Persistent Neighborhood Communication on Process Topologies 8.8 18 19 Persistent variants of the neighborhood collective operations can offer significant perfor-20mance benefits for programs with repetitive communication patterns. The semantics are 21similar to persistent collective operations as described in Section 6.13. 22 238.8.1 Persistent Neighborhood Gather 24 2526MPI_NEIGHBOR_ALLGATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, 27recvtype, comm, info, request) 2829 sendbuf IN starting address of send buffer (choice) 30 IN sendcount number of elements sent to each neighbor 31 (non-negative integer) 32 IN sendtype data type of send buffer elements (handle) 33 34 OUT recvbuf starting address of receive buffer (choice) 35 IN recvcount number of elements received from each neighbor 36 (non-negative integer) 37 IN recvtype data type of receive buffer elements (handle) 38 39 IN comm communicator with topology structure (handle) 40 IN info info argument (handle) 41 OUT request communication request (handle) 4243 44C binding 45int MPI_Neighbor_allgather_init(const void *sendbuf, int sendcount, 46MPI_Datatype sendtype, void *recvbuf, int recvcount, 47 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 48 MPI_Request *request)

Fortran	2008 binding	
	0	(sendbuf, sendcount, sendtype, recvbuf,
		cvtype, comm, info, request, ierror)
		INTENT(IN), ASYNCHRONOUS :: sendbuf
		sendcount, recvcount
	• 1	ENT(IN) :: sendtype, recvtype ASYNCHRONOUS :: recvbuf
	E(#), DIMENSION(), E(MPI_Comm), INTENT(
	E(MPI_Info), INTENT(
	E(MPI_Request), INTE	
INTE	EGER, OPTIONAL, INTE	ENT(OUT) :: ierror
ortran	binding	
	•	C(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
_		CVTYPE, COMM, INFO, REQUEST, IERROR)
	<pre>> SENDBUF(*), RECV</pre>	
INTE		DTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
	IERROR	
Crea	tes a persistent collect	ive communication request for the neighborhood allgather
peration		
		/_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
		comm, info, request)
IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each neighbor
		(non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length indegree)
		containing the number of elements that are received
		from each neighbor
IN	displs	integer array (of length indegree). Entry i specifies
		the displacement (relative to recvbuf) at which to
		place the incoming data from neighbor i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)
bindi	ng	
	0	<pre>v_init(const void *sendbuf, int sendcount,</pre>
	0	<pre>sendtype, void *recvbuf, const int recvcounts[],</pre>
		pls[], MPI_Datatype recvtype, MPI_Comm comm,
		MDT Derwest (mercent)

Unofficial Draft for Comment Only

MPI_Info info, MPI_Request *request)

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
3
                    recvcounts, displs, recvtype, comm, info, request, ierror)
4
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
          INTEGER, INTENT(IN) :: sendcount, displs(*)
6
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
8
          INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
9
          TYPE(MPI_Comm), INTENT(IN) :: comm
10
          TYPE(MPI_Info), INTENT(IN) :: info
11
          TYPE(MPI_Request), INTENT(OUT) :: request
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     Fortran binding
14
     MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
15
                    RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
16
          <type> SENDBUF(*), RECVBUF(*)
17
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
18
                     INFO, REQUEST, IERROR
19
20
          Creates a persistent collective communication request for the neighborhood allgathery
21
     operation.
22
23
     8.8.2 Persistent Neighborhood Alltoall
24
25
26
     MPI_NEIGHBOR_ALLTOALL_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount,
27
                    recvtype, comm, info, request)
28
       IN
                 sendbuf
                                             starting address of send buffer (choice)
29
30
       IN
                 sendcount
                                             number of elements sent to each neighbor
31
                                             (non-negative integer)
32
       IN
                 sendtype
                                             data type of send buffer elements (handle)
33
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
34
35
       ĬN
                 recvcount
                                             number of elements received from each neighbor
36
                                             (non-negative integer)
37
                 recvtype
                                             data type of receive buffer elements (handle)
       IN
38
       IN
                 comm
                                             communicator with topology structure (handle)
39
40
                 info
       IN
                                             info argument (handle)
41
       OUT
                                             communication request (handle)
                 request
42
43
     C binding
44
     int MPI_Neighbor_alltoall_init(const void *sendbuf, int sendcount,
45
                    MPI_Datatype sendtype, void *recvbuf, int recvcount,
46
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
47
                    MPI_Request *request)
48
```

	n 2008 binding		1		
MPI_Nei	•	it(sendbuf, sendcount, sendtype, recvbuf,	2		
		recvtype, comm, info, request, ierror)	3		
		.), INTENT(IN), ASYNCHRONOUS :: sendbuf	4		
	-	:: sendcount, recvcount	5 6		
	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf				
	PE(MPI_Comm), INTE		7 8		
	PE(MPI_Info), INTE		9		
		NTENT(OUT) :: request	10		
	-	NTENT(OUT) :: ierror	11		
			12		
	n binding		13		
MPI_NE		IT (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,	14		
<+	vpe> SENDBUF(*), R	RECVTYPE, COMM, INFO, REQUEST, IERROR)	15		
	•	ENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,	16		
INI	IERROR		17		
a			18 19		
	-	lective communication request for the neighborhood alltoall	20		
operatio	n.		20		
			22		
MPI_NE	GHBOR_ALLTOALL	V_INIT(sendbuf, sendcounts, sdispls, sendtype, recvbuf,	23		
	recvcounts, ro	lispls, recvtype, comm, info, request)	24		
IN	sendbuf	starting address of send buffer (choice)	25		
IN	sendcounts	non-negative integer array (of length outdegree)	26		
	Schucounts	specifying the number of elements to send to each	27		
		neighbor	28		
IN	sdispls	integer array (of length outdegree). Entry j specifies	29		
	5015015	the displacement (relative to sendbuf) from which	30 31		
		send the outgoing data to neighbor j	32		
IN	sendtype	data type of send buffer elements (handle)	33		
OUT	recvbuf	starting address of receive buffer (choice)	34		
			35		
IN	recvcounts	non-negative integer array (of length indegree)	36		
		specifying the number of elements that are received from each neighbor	37		
		-	38		
IN	rdispls	integer array (of length indegree). Entry i specifies	39		
		the displacement (relative to recvbuf) at which to	40		
		place the incoming data from neighbor i	41		
IN	recvtype	data type of receive buffer elements (handle)	42 43		
IN	comm	communicator with topology structure (handle)	43 44		
IN	info	info argument (handle)	45		
OUT	request	communication request (handle)	46		
			47		
C bind	ing		48		

C binding

```
1
     int MPI_Neighbor_alltoallv_init(const void *sendbuf,
\mathbf{2}
                   const int sendcounts[], const int sdispls[],
3
                   MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
4
                   const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm,
5
                   MPI_Info info, MPI_Request *request)
6
     Fortran 2008 binding
7
     MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
8
                   recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
9
                   ierror)
10
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
11
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
12
                    recvcounts(*), rdispls(*)
13
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
14
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
15
         TYPE(MPI_Comm), INTENT(IN) :: comm
16
         TYPE(MPI_Info), INTENT(IN) :: info
17
         TYPE(MPI_Request), INTENT(OUT) :: request
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     Fortran binding
21
     MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
22
                   RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST,
23
                   IERROR)
24
         <type> SENDBUF(*), RECVBUF(*)
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
25
26
                    RECVTYPE, COMM, INFO, REQUEST, IERROR
27
         Creates a persistent collective communication request for the neighborhood alloally
28
     operation.
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

MPI_NEI		NIT(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, s, recvtypes, comm, info, request)	1 2
IN	sendbuf	starting address of send buffer (choice)	3
			4
IN	sendcounts	non-negative integer array (of length outdegree)	5
		specifying the number of elements to send to each neighbor	6
	adian la		7 8
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from	9
		which to take the outgoing data destined for	10
		neighbor j (array of integers)	11
IN	sendtypes	array of datatypes (of length outdegree). Entry j	12
	Senacypes	specifies the type of data to send to neighbor j (array	13
		of handles)	14
Ουτ	recvbuf	starting address of receive buffer (choice)	15 16
IN	recvcounts	non-negative integer array (of length indegree)	17
		specifying the number of elements that are received	18
		from each neighbor	19
IN	rdispls	integer array (of length indegree). Entry i specifies	20
		the displacement in bytes (relative to recvbuf) at	21
		which to place the incoming data from neighbor i	22 23
		(array of integers)	23 24
IN	recvtypes	array of datatypes (of length indegree). Entry i	25
		specifies the type of data received from neighbor i	26
		(array of handles)	27
IN	comm	communicator with topology structure (handle)	28
IN	info	info argument (handle)	29 30
OUT	request	communication request (handle)	31
		/	32
C bindi	ng		33
int MPI	Ŭ,	init(const void *sendbuf,	34
		<pre>lcounts[], const MPI_Aint sdispls[],</pre>	35
		type sendtypes[], void *recvbuf,	36
		<pre>vcounts[], const MPI_Aint rdispls[], utype recvtypes[], MPI_Comm comm, MPI_Info info,</pre>	37 38
	MPI_Request *r		39
D (_		40
	2008 binding	(and but condecurts adian) a conditions	41
MPI_Nel	-	(sendbuf, sendcounts, sdispls, sendtypes, counts, rdispls, recvtypes, comm, info, request,	42
	ierror)	Jourdo, Turopio, Toovoypoo, Journa, Into, Toquobo,	43
TYP		INTENT(IN), ASYNCHRONOUS :: sendbuf	44
INT	EGER, INTENT(IN), AS	YNCHRONOUS :: sendcounts(*), recvcounts(*)	45 46
INT		<pre>S_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),</pre>	40
	rdispls(*)		48

```
1
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
2
                    recvtypes(*)
3
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
4
         TYPE(MPI_Comm), INTENT(IN) :: comm
5
         TYPE(MPI_Info), INTENT(IN) :: info
6
         TYPE(MPI_Request), INTENT(OUT) :: request
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     Fortran binding
9
     MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES,
10
                   RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST,
11
                   IERROR)
12
         <type> SENDBUF(*), RECVBUF(*)
13
         INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
14
                    INFO, REQUEST, IERROR
15
         INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
16
17
         Creates a persistent collective communication request for the neighborhood alltoallw
```

operation.

8.9 An Application Example

 24

Example 8.11 The example in Figures 8.4-8.7 shows how the grid definition and inquiry functions can be used in an application program. A partial differential equation, for instance the Poisson equation, is to be solved on a rectangular domain. First, the processes organize themselves in a two-dimensional structure. Each process then inquires about the ranks of its neighbors in the four directions (up, down, right, left). The numerical problem is solved by an iterative method, the details of which are hidden in the subroutine relax.

In each relaxation step each process computes new values for the solution grid function at the points u(1:100,1:100) owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the newly calculated values in u(1,1:100) must be sent into the halo cells u(101,1:100) of the left-hand neighbor with coordinates (own_coord(1)-1,own_coord(2)).

```
INTEGER ndims, num_neigh
                                                                                     1
LOGICAL reorder
                                                                                     2
PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
INTEGER comm, comm_size, comm_cart, dims(ndims), ierr
INTEGER neigh_rank(num_neigh), own_coords(ndims), i, j, it
                                                                                     5
LOGICAL periods(ndims)
                                                                                     6
REAL u(0:101,0:101), f(0:101,0:101)
DATA dims / ndims * 0 /
comm = MPI_COMM_WORLD
CALL MPI_COMM_SIZE(comm, comm_size, ierr)
                                                                                     10
    Set process grid size and periodicity
!
                                                                                     11
CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
                                                                                     12
periods(1) = .TRUE.
                                                                                     13
periods(2) = .TRUE.
                                                                                     14
    Create a grid structure in WORLD group and inquire about own position
1
                                                                                     15
CALL MPI_CART_CREATE(comm, ndims, dims, periods, reorder, &
                                                                                     16
                      comm_cart, ierr)
                                                                                     17
CALL MPI_CART_GET(comm_cart, ndims, dims, periods, own_coords, ierr)
                                                                                     18
i = own_coords(1)
                                                                                     19
j = own_coords(2)
                                                                                     20
! Look up the ranks for the neighbors. Own process coordinates are (i,j).
                                                                                    21
! Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
                                                                                    22
CALL MPI_CART_SHIFT(comm_cart, 0,1, neigh_rank(1), neigh_rank(2), ierr)
                                                                                    23
CALL MPI_CART_SHIFT(comm_cart, 1,1, neigh_rank(3), neigh_rank(4), ierr)
                                                                                     24
! Initialize the grid functions and start the iteration
                                                                                     25
CALL init(u, f)
                                                                                     26
DO it=1,100
                                                                                     27
   CALL relax(u, f)
                                                                                     28
       Exchange data with neighbor processes
!
                                                                                     29
   CALL exchange(u, comm_cart, neigh_rank, num_neigh)
                                                                                     30
END DO
                                                                                     31
CALL output(u)
                                                                                     32
                                                                                     33
                                                                                     34
  Figure 8.4: Set-up of process structure for two-dimensional parallel Poisson solver.
                                                                                     35
                                                                                     36
                                                                                     37
                                                                                     38
                                                                                     39
                                                                                     40
                                                                                     41
                                                                                     42
                                                                                     43
```

```
SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
1
     REAL u(0:101,0:101)
\mathbf{2}
     INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
3
     REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
4
     INTEGER ierr
5
     sndbuf(1:100,1) = u(1,1:100)
6
     sndbuf(1:100,2) = u(100,1:100)
7
     sndbuf(1:100,3) = u(1:100, 1)
8
     sndbuf(1:100,4) = u(1:100,100)
9
     CALL MPI_NEIGHBOR_ALLTOALL(sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
10
                                  comm_cart, ierr)
11
     ! instead of
12
     ! CALL MPI_IRECV(rcvbuf(1,1),100,MPI_REAL, neigh_rank(1),..., rq(1), ierr)
13
     ! CALL MPI_ISEND(sndbuf(1,2),100,MPI_REAL, neigh_rank(2),..., rq(2), ierr)
14
     Т
         Always pairing a receive from rank_source with a send to rank_dest
15
         of the same direction in MPI_CART_SHIFT!
     Т
16
     ! CALL MPI_IRECV(rcvbuf(1,2),100,MPI_REAL, neigh_rank(2),..., rq(3), ierr)
17
     ! CALL MPI_ISEND(sndbuf(1,1),100,MPI_REAL, neigh_rank(1),..., rq(4), ierr)
18
     ! CALL MPI_IRECV(rcvbuf(1,3),100,MPI_REAL, neigh_rank(3),..., rq(5), ierr)
19
     ! CALL MPI_ISEND(sndbuf(1,4),100,MPI_REAL, neigh_rank(4),..., rq(6), ierr)
20
     ! CALL MPI_IRECV(rcvbuf(1,4),100,MPI_REAL, neigh_rank(4),..., rq(7), ierr)
21
     ! CALL MPI_ISEND(sndbuf(1,3),100,MPI_REAL, neigh_rank(3),..., rq(8), ierr)
22
         Of course, one can first start all four IRECV and then all four ISEND,
     1
23
         Or vice versa, but both in the sequence shown above. Otherwise, the
     1
24
         matching would be wrong for 2 or only 1 processes in a direction.
     !
25
     ! CALL MPI_WAITALL(2*num_neigh, rq, statuses, ierr)
26
     u(0,1:100) = rcvbuf(1:100,1)
27
     u(101,1:100) = rcvbuf(1:100,2)
28
     u(1:100, 0) = rcvbuf(1:100,3)
29
     u(1:100,101) = rcvbuf(1:100,4)
30
     END
^{31}
32
33
     Figure 8.5: Communication routine with local data copying and sparse neighborhood all-
34
     to-all.
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

```
SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
                                                                                          1
IMPLICIT NONE
                                                                                          2
USE MPI
REAL u(0:101,0:101)
                                                                                          4
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
                                                                                          5
INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
                                                                                          6
INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
INTEGER (KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
INTEGER type_vec, ierr
                                                                                          9
! The following initialization need to be done only once
                                                                                          10
! before the first call of exchange.
                                                                                          11
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
                                                                                          12
CALL MPI_TYPE_VECTOR(100, 1, 102, MPI_REAL, type_vec, ierr)
                                                                                          13
CALL MPI_TYPE_COMMIT(type_vec, ierr)
sndtypes(1:2) = type_vec
                                                                                          14
sndcounts(1:2) = 1
                                                                                          15
sndtypes(3:4) = MPI_REAL
                                                                                          16
sndcounts(3:4) = 100
                                                                                          17
rcvtypes = sndtypes
                                                                                          18
rcvcounts = sndcounts
                                                                                          19
sdispls(1) = ( 1 + 1*102) * sizeofreal ! first element of u( 1
                                                                           1:100)
                                                                                          20
                      1*102) * sizeofreal ! first element of u(100
sdispls(2) = (100 +
                                                                           1:100
                                                                                          21
sdispls(3) = ( 1 +
                     1*102) * sizeofreal ! first element of u( 1:100,
                                                                                 )
                                                                           1
sdispls(4) = (1 + 100*102) * size of real ! first element of u(
                                                                  1:100,100
                                                                                 )
                                                                                          22
rdispls(1) = (0 +
                     1*102) * sizeofreal ! first element of u( 0
                                                                           1:100)
                                                                         .
                                                                                          23
rdispls(2) = (101 +
                      1*102) * sizeofreal ! first element of u(101
                                                                           1:100)
                                                                                          24
rdispls(3) = ( 1 +
                      0*102) * sizeofreal ! first element of u( 1:100, 0
                                                                                 )
                                                                                          25
rdispls(4) = (1 + 101*102) * sizeofreal ! first element of u( 1:100,101
                                                                                 )
                                                                                          26
! the following communication has to be done in each call of exchange
                                                                                          27
CALL MPI_NEIGHBOR_ALLTOALLW(u, sndcounts, sdispls, sndtypes, &
                                                                                          28
                             u, rcvcounts, rdispls, rcvtypes, &
                                                                                          29
                             comm_cart, ierr)
! The following finalizing need to be done only once
                                                                                          30
! after the last call of exchange.
                                                                                          31
CALL MPI_TYPE_FREE(type_vec, ierr)
                                                                                          32
END
                                                                                          33
                                                                                          34
                                                                                          35
Figure 8.6: Communication routine with sparse neighborhood all-to-all-w and without local
                                                                                          36
data copying.
                                                                                          37
                                                                                          38
                                                                                          39
                                                                                          40
                                                                                          41
                                                                                          42
                                                                                          43
                                                                                          44
                                                                                          45
                                                                                          46
                                                                                          47
```

```
INTEGER ndims, num_neigh
1
     LOGICAL reorder
2
    PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
3
     INTEGER comm, comm_size, comm_cart, dims(ndims), it, ierr
4
     LOGICAL periods(ndims)
5
    REAL u(0:101,0:101), f(0:101,0:101)
6
    DATA dims / ndims * 0 /
7
     INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
8
     INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
9
     INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
10
     INTEGER (KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
11
     INTEGER type_vec, request, status
12
     comm = MPI_COMM_WORLD
13
     CALL MPI_COMM_SIZE(comm, comm_size, ierr)
14
         Set process grid size and periodicity
15
     CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
16
     periods(1) = .TRUE.
17
     periods(2) = .TRUE.
18
         Create a grid structure in WORLD group
     !
19
     CALL MPI_CART_CREATE(comm, ndims, dims, periods, reorder, &
20
                           comm_cart, ierr)
21
     ! Create datatypes for the neighborhood communication
22
     1
23
     ! Insert code from example in Figure 7.4 to create and initialize
24
     ! sndcounts, sdispls, sndtypes, rcvcounts, rdispls, and rcvtypes
25
     Ţ
26
     ! Initialize the neighborhood all-to-all-w operation
27
     CALL MPI_NEIGHBOR_ALLTOALLW_INIT(u, sndcounts, sdispls, sndtypes, &
28
                                        u, rcvcounts, rdispls, rcvtypes, &
29
                                        comm_cart, info, request, ierr)
30
     ! Initialize the grid functions and start the iteration
31
     CALL init(u, f)
32
     DO it=1,100
33
            Start data exchange with neighbor processes
     34
       CALL MPI_START(request, ierr)
35
            Compute inner cells
     !
36
        CALL relax_inner (u, f)
37
            Check on completion of neighbor exchange
     !
38
        CALL MPI_WAIT(request, status, ierr)
39
            Compute edge cells
     1
40
        CALL relax_edges(u, f)
41
     END DO
42
     CALL output(u)
43
     CALL MPI_REQUEST_FREE(request, ierr)
44
     CALL MPI_TYPE_FREE(type_vec, ierr)
45
46
47
     Figure 8.7: Two-dimensional parallel Poisson solver with persistent sparse neighborhood
```

⁴⁸ all-to-all-w and without local data copying.

Unofficial Draft for Comment Only

Chapter 9

MPI Environmental Management

 $\frac{24}{25}$

 $41 \\ 42$

This chapter discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation and the execution environment (such as error handling). The procedures for entering and leaving the MPI execution environment are also described here.

9.1 Implementation Information

9.1.1 Version Inquiries

In order to cope with changes to the MPI Standard, there are both compile-time and runtime ways to determine which version of the standard is in use in the environment one is using.

The "version" will be represented by two separate integers, for the version and subversion: In C,

```
#define MPI_VERSION 3
#define MPI_SUBVERSION 1
```

in Fortran,

```
INTEGER :: MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION = 3)
PARAMETER (MPI_SUBVERSION = 1)
```

For runtime determination,

MPI_GET_VERSION(version, subversion)

OUT	version	version number (integer)
OUT	subversion	subversion number (integer)

Сb	inding	S				
int	MPI_G	et_	version(int	*version,	int	*subversion)

```
Fortran 2008 binding
```

```
MPI_Get_version(version, subversion, ierror)
```

```
1
          INTEGER, INTENT(OUT) :: version, subversion
\mathbf{2}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     Fortran binding
4
     MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
5
          INTEGER VERSION, SUBVERSION, IERROR
6
7
          MPI_GET_VERSION can be called at any time in an MPI program. This function must
8
     always be thread-safe, as defined in Section 11.6. Valid (MPI_VERSION, MPI_SUBVERSION)
9
     pairs in this and previous versions of the MPI standard are (4,0), (3,1), (3,0), (2,2), (2,1),
10
     (2,0), and (1,2).
11
12
     MPI_GET_LIBRARY_VERSION(version, resultlen)
13
14
       OUT
                 version
                                              version number (string)
15
       OUT
                 resultlen
                                              Length (in printable characters) of the result
16
                                              returned in version (integer)
17
18
     C binding
19
     int MPI_Get_library_version(char *version, int *resultlen)
20
21
     Fortran 2008 binding
22
     MPI_Get_library_version(version, resultlen, ierror)
23
          CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version
^{24}
          INTEGER, INTENT(OUT) :: resultlen
25
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     Fortran binding
27
     MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR)
28
          CHARACTER*(*) VERSION
29
          INTEGER RESULTLEN, IERROR
30
^{31}
          This routine returns a string representing the version of the MPI library. The version
32
     argument is a character string for maximum flexibility.
33
34
           Advice to implementors. An implementation of MPI should return a different string
35
           for every change to its source code or build that could be visible to the user. (End of
36
           advice to implementors.)
37
38
          The argument version must represent storage that is
39
     MPI_MAX_LIBRARY_VERSION_STRING characters long. MPI_GET_LIBRARY_VERSION may
40
     write up to this many characters into version.
41
          The number of characters actually written is returned in the output argument, resultlen.
42
     In C, a null character is additionally stored at version[resultlen]. The value of resultlen cannot
43
     be larger than MPI_MAX_LIBRARY_VERSION_STRING - 1. In Fortran, version is padded on
44
     the right with blank characters. The value of resultlen cannot be larger than
45
     MPI_MAX_LIBRARY_VERSION_STRING.
46
          MPI_GET_LIBRARY_VERSION can be called at any time in an MPI program. This
47
     function must always be thread-safe, as defined in Section 11.6.
```

CHAPTER 9. MPI ENVIRONMENTAL MANAGEMENT

48

1
2 3
4
5
6
7
8
9
10 11
12
13
14
15
16
17 18
19
20
20
22
23
24
25
26
27
28
29
30
31
32
33
34
35
Ů₽̂S,
37
38
39
40
41
42
43
44
45
46
47
48

1	IO Rank		
2 3	The value re	eturned for MPI_IO is the ranl	k of a processor that can provide language-standard
4			hat all of the Fortran I/O operations are supported
5	()		s means that all of the ISO C I/O operations are
6	、	e.g., fopen, fprintf, lseek	
7	-		ge-standard I/O, then the value MPI_ANY_SOURCE
8			lling process can provide language-standard I/O, se, if some process can provide language-standard
9			s will be returned. The same value need not be
10 11	,	-	can provide language-standard I/O, then the value
12	MPI_PROC_	NULL will be returned.	
13	4 -1	Netethet inset	
14 15			s not collective, and this attribute does <i>not</i> indicate input. (<i>End of advice to users.</i>)
16	Clash Const		
17	Clock Synch		
18 19			GLOBAL is 1 if clocks at all processes in
20		, ,	0 otherwise. A collection of clocks is considered aken to synchronize them. The expectation is that
21	-	_	alls to MPI_WTIME, will be less then one half the
22			length zero. If time is measured at a process just
23			st after a matching receive, the second time should
24 25	•	igher than the first one.	
26			BAL need not be present when the clocks are not ey MPI_WTIME_IS_GLOBAL is always valid). This
27	-		unicators other then MPI_COMM_WORLD.
28		-	BAL has the same value on all processes of
29	MPI_COMM	_WORLD.	
30 31			
32	Inquire Proc	essor Name	
33			
34	MPL GET F	PROCESSOR_NAME(name, r	esultlen)
35	OUT	name	A unique specifier for the actual (as opposed to
36 37	001	hame	virtual) node.
38	OUT	resultlen	Length (in printable characters) of the result
39			returned in name
40			
41 42	C binding		
43	int MPI_Ge	et_processor_name(char *)	name, int *resultlen)
44	Fortran 20	008 binding	
45	-	cocessor_name(name, resul	
46 47		CTER(LEN=MPI_MAX_PROCESS(ER, INTENT(OUT) :: result	DR_NAME), INTENT(OUT) :: name
47 48		ER, OPTIONAL, INTENT(OUT)	

Unofficial Draft for Comment Only

Fortran binding MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR) CHARACTER*(*) NAME INTEGER RESULTLEN, IERROR

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the running homogeneous system). The argument name must represent storage that is at least MPI_MAX_PROCESSOR_NAME characters long. MPI_GET_PROCESSOR_NAME may write up to this many characters into name.

The number of characters actually written is returned in the output argument, resultlen. In C, a null character is additionally stored at name[resultlen]. The value of resultlen cannot be larger than MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. The value of resultlen cannot be larger than MPI_MAX_PROCESSOR_NAME.

Rationale. This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI *requires* or defines process migration; this definition of MPI_GET_PROCESSOR_NAME simply allows such an implementation. (*End of rationale.*)

Advice to users. The user must provide at least MPI_MAX_PROCESSOR_NAME space to write the processor name — processor names can be this long. The user should examine the output argument, resultlen, to determine the actual length of the name. (End of advice to users.)

9.2 Memory Allocation

In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of some RMA functionality as defined in Section 12.5.3.

MPI_ALLOC_MEM(size, info, baseptr)

IN	size	size of memory segment in bytes (non-negative	40
		$\operatorname{integer})$	41
IN	info	info argument (handle)	42
OUT	baseptr	pointer to beginning of memory segment allocated	43
001		Pointer to seguring of memory pegment anotated	44

C binding int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)

Fortran 2008 binding

1

 $\mathbf{2}$

3

4 5

6

7

8

9

10

11

12

13

14

15

16 17

18

19

20

21 22

23

 24

25

26 27

28 29

30

 31

32

33

34

35

36 37 38

39

45

46

```
1
     MPI_Alloc_mem(size, info, baseptr, ierror)
\mathbf{2}
          USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
3
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
4
          TYPE(MPI_Info), INTENT(IN) :: info
5
          TYPE(C_PTR), INTENT(OUT) :: baseptr
6
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     Fortran binding
8
     MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
9
          INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
10
          INTEGER INFO, IERROR
11
12
         If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must
13
     be provided in the mpi module and should be provided in mpif.h through overloading,
14
     i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
15
     BASEPTR, but with a different specific procedure name:
16
17
     INTERFACE MPI_ALLOC_MEM
18
          SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
19
              IMPORT :: MPI_ADDRESS_KIND
              INTEGER INFO, IERROR
20
              INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
21
22
          END SUBROUTINE
23
          SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
^{24}
              USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
25
              IMPORT :: MPI_ADDRESS_KIND
26
              INTEGER :: INFO, IERROR
27
              INTEGER(KIND=MPI_ADDRESS_KIND) ::
                                                      SIZE
              TYPE(C_PTR) ::
                                BASEPTR
28
29
          END SUBROUTINE
30
     END INTERFACE
31
          The base procedure name of this overloaded function is MPI_ALLOC_MEM_CPTR. The
32
     implied specific procedure names are described in Section 19.1.5.
33
          By default, the allocated memory shall be aligned to at least the alignment required
34
     for load/store accesses of any datatype corresponding to a predefined MPI datatype. The
35
     info argument may be used to specify a desired alternative minimum alignment in bytes for
36
     the allocated memory by setting the value of the key "mpi_minimum_memory_alignment" to an
37
     integral number equal to a power of two. An implementation may ignore values smaller than
38
     the default required alignment. The info argument can also be used to provide directives
39
     that control the desired location of the allocated memory. Such a directive does not affect
40
     the semantics of the call. The corresponding info values are implementation-dependent. A
41
     null directive value of info = MPI_INFO_NULL is always valid.
42
          The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM
43
     to indicate it failed because memory is exhausted.
44
45
46
47
48
```

MPI_FREI	E_MEM(base)	
IN	base	initial address of memory segment allocated by MPI_ALLOC_MEM (choice)
C bindin int MPI_1	g Free_mem(void *base)	
MPI_Free TYPE	2008 binding _mem(base, ierror) (*), DIMENSION(), INTEN GER, OPTIONAL, INTENT(OUT	T(IN), ASYNCHRONOUS :: base) :: ierror
<type< td=""><td>binding _MEM(BASE, IERROR) e> BASE(*) GER IERROR</td><td></td></type<>	binding _MEM(BASE, IERROR) e> BASE(*) GER IERROR	

The function MPI_FREE_MEM may return an error code of class MPI_ERR_BASE to indicate an invalid base argument.

Rationale. The C bindings of MPI_ALLOC_MEM and MPI_FREE_MEM are similar to the bindings for the malloc and free C library calls: a call to MPI_Alloc_mem(..., &base) should be paired with a call to MPI_Free_mem(base) (one less level of indirection). Both arguments are declared to be of same type void* so as to facilitate type casting. The Fortran binding is consistent with the C bindings: the Fortran MPI_ALLOC_MEM call returns in baseptr the TYPE(C_PTR) pointer or the (integer valued) address of the allocated memory. The base argument of MPI_FREE_MEM is a choice argument, which passes (a reference to) the variable stored at that location. (End of rationale.)

If MPI_ALLOC_MEM allocates special memory, then a Advice to implementors. design similar to the design of C malloc and free functions has to be used, in order to find out the size of a memory segment, when the segment is freed. If no special memory is used, MPI_ALLOC_MEM simply invokes malloc, and MPI_FREE_MEM invokes free.

A call to MPI_ALLOC_MEM can be used in shared memory systems to allocate memory in a shared memory segment. (End of advice to implementors.)

Example 9.1 Example of use of MPI_ALLOC_MEM, in Fortran with TYPE(C_PTR) pointers. We assume 4-byte REALs.

USE mpi_f08 ! or USE mpi (not guaranteed with INCLUDE 'mpif.h') USE, INTRINSIC :: ISO_C_BINDING TYPE(C_PTR) :: p REAL, DIMENSION(:,:), POINTER :: a ! no memory is allocated INTEGER, DIMENSION(2) :: shape INTEGER(KIND=MPI_ADDRESS_KIND) :: size shape = (/100, 100/)

1 $\mathbf{2}$

3 4 5

6

9

10

11

12

13

14

15

1617

18

19

2021

22 23

24

25

26

2728

29

30

31

32

33

34 35

36

37 38

39

40 41

42

43

44

45

46

47

```
1
       size = 4 * \text{shape}(1) * \text{shape}(2)
                                                             ! assuming 4 bytes per REAL
2
       CALL MPI_Alloc_mem(size, MPI_INFO_NULL, p, ierr) ! memory is allocated and
3
       CALL C_F_POINTER(p, a, shape) ! intrinsic
                                                             ! now accessible via a(i,j)
4
                                          ! in ISO_C_BINDING
        . . .
5
       a(3,5) = 2.71
6
        . . .
7
       CALL MPI_Free_mem(a, ierr)
                                                             ! memory is freed
8
9
     Example 9.2 Example of use of MPI_ALLOC_MEM, in Fortran with non-standard Cray-
10
     pointers. We assume 4-byte REALS, and assume that these pointers are address-sized.
11
12
       REAL A
13
                                      ! no memory is allocated
       POINTER (P, A(100,100))
14
       INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
15
       SIZE = 4*100*100
16
       CALL MPI_ALLOC_MEM(SIZE, MPI_INFO_NULL, P, IERR)
17
        ! memory is allocated
18
        . . .
19
       A(3,5) = 2.71
20
        . . .
21
       CALL MPI_FREE_MEM(A, IERR) ! memory is freed
22
         This code is not Fortran 77 or Fortran 90 code. Some compilers may not support this
23
     code or need a special option, e.g., the GNU gFortran compiler needs -fcray-pointer.
24
25
           Advice to implementors. Some compilers map Cray-pointers to address-sized integers,
26
           some to TYPE(C_PTR) pointers (e.g., Cray Fortran, version 7.3.3). From the user's
27
           viewpoint, this mapping is irrelevant because Examples 9.2 should work correctly
28
           with an MPI-3.0 (or later) library if Cray-pointers are available. (End of advice to
29
           implementors.)
30
31
32
     Example 9.3 Same example, in C.
33
       float (* f)[100][100];
34
       /* no memory is allocated */
35
       MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
36
       /* memory allocated */
37
        . . .
38
        (*f)[5][3] = 2.71;
39
        . . .
40
       MPI_Free_mem(f);
41
42
43
            Error Handling
     9.3
44
```

An MPI implementation may be unable or choose not to handle some failures that occur
 during MPI calls. These can include failures that generate exceptions or traps, such as
 floating point errors or access violations. The set of failures that are handled by MPI is
 implementation-dependent. Each such failure causes an error to be raised.

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8 9

10

11

12

13

14

15

16

17 18

19

20

21

22

23

 24

25 26

27 28

29

30

 31

32 33

34

35

The above text takes precedence over any text on error handling within this document. Specifically, text that states that errors *will* be handled should be read as *may* be handled. More background information about how MPI treats errors can be found in Section 2.8.

A user can associate error handlers to four types of objects: communicators, windows, files, and sessions. The specified error handling routine will be used for any error that occurs during a call to MPI for the respective object. MPI calls that are not related to any MPI objects are considered to be attached to the communicator MPI_COMM_SELF. When MPI_COMM_SELF is not initialized (i.e., before MPI_INIT / MPI_INIT_THREAD or after MPI_FINALIZE) the error raises the initial error handler (set during the launch operation, see 11.8.4). The attachment of error handlers to objects is purely local: different processes may attach different error handlers to corresponding objects.

Several predefined error handlers are available in MPI:

- MPI_ERRORS_ARE_FATAL The handler, when called, causes the program to abort all connected MPI processes. This is similar to calling MPI_ABORT using a communicator containing all connected processes with an implementation-specific value as the errorcode argument.
- **MPI_ERRORS_ABORT** The handler, when called, is invoked on a communicator in a manner similar to calling MPI_ABORT on that communicator. If the error handler is invoked on an window or file, it is similar to calling MPI_ABORT using a communicator containing the group of MPI processes associated with the window or file, respectively. If the error handler is invoked on a session, the operation aborts only the local MPI process. In all cases, the value that would be provided as the errorcode argument to MPI_ABORT is implementation-specific.
- **MPI_ERRORS_RETURN** The handler has no effect other than returning the error code to the user.

Advice to implementors. The implementation-specific error information resulting from MPI_ERRORS_ARE_FATAL and MPI_ERRORS_ABORT provided to the invoking environment should be meaningful to the end-user, for example a predefined error class. (End of advice to implementors.)

Implementations may provide additional predefined error handlers and programmers can code their own error handlers.

Unless otherwise requested, the error handler MPI_ERRORS_ARE_FATAL is set as the 36 default initial error handler and associated with predefined communicators. Thus, if the 37 user chooses not to control error handling, every error that MPI handles is treated as fatal. 38 Since (almost) all MPI calls return an error code, a user may choose to handle errors in its 39 main code, by testing the return code of MPI calls and executing a suitable recovery code 40 when the call was not successful. In this case, the error handler MPI_ERRORS_RETURN will 41 be used. Usually it is more convenient and more efficient not to test for errors after each 42MPI call, and have such error handled by a non-trivial MPI error handler. Note that unlike 43 predefined communicators, windows and files do not inherit from the initial error handler, 44as defined in Sections 12.6 and 14.7 respectively. 45

When an error is raised, MPI will provide the user information about that error using an error code. Some errors might prevent MPI from completing further API calls successfully and those functions will continue to report errors until the cause of the error is corrected 48

 $\mathbf{2}$

3

4

5

6 7

8

9

10

11

12 13 or the user terminates the application. The user can make the determination of whether or not to attempt to continue when handling such an error.

Advice to users. For example, users may be unable to correct errors corresponding to some error classes, such as MPI_ERR_INTERN. Such errors may cause subsequent MPI calls to complete in error. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors and available recovery actions. (End of advice to implementors.)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are provided to create new error handlers, to associate error handlers with objects, and to test which error handler is associated with an object. C has distinct typedefs for user defined error handling callback functions that accept communicator, file, window, and session arguments. In Fortran there are four user routines.

¹⁹ An error handler object is created by a call to MPI_XXX_CREATE_ERRHANDLER, ²⁰ where XXX is, respectively, COMM, WIN, FILE, or SESSION.

An error handler is attached to a communicator, window, file, or session by a call to MPI_XXX_SET_ERRHANDLER. The error handler must be either a predefined error handler, or an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER, with matching XXX. An error handler can also be attached to a session using the errorhandler argument to MPI_SESSION_INIT. The predefined error handlers MPI_ERRORS_RETURN and MPI_ERRORS_ARE_FATAL can be attached to communicators, windows, files, or sessions.

The error handler currently associated with a communicator, window, file, or session can be retrieved by a call to MPI_XXX_GET_ERRHANDLER.

The MPI function MPI_ERRHANDLER_FREE can be used to free an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER.

³¹ MPI_XXX_GET_ERRHANDLER behave as if a new error handler object is created. That ³² is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called ³³ with the error handler returned from MPI_XXX_GET_ERRHANDLER to mark the error ³⁴ handler for deallocation. This provides behavior similar to that of MPI_COMM_GROUP ³⁵ and MPI_GROUP_FREE.

Advice to implementors. High-quality implementations should raise an error when an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER is attached to an object of the wrong type with a call to MPI_YYY_SET_ERRHANDLER. To do so, it is necessary to maintain, with each error handler, information on the typedef of the associated user function. (*End of advice to implementors.*)

The syntax for these calls is given below.

43 44

36

37

38

39

40

- 45
- 46
- 47 48

9.3.	ERROR H	ANDLING		417
9.3.	1 Error Ha	ndlers for Communicate	Drs	1
				2
				4
			omm_errhandler_fn, errhandler)	5
IN	com	m_errhandler_fn	user defined error handling procedure (function)	6
Ol	JT errh	andler	MPI error handler (handle)	7
				8
	inding			9 10
int		create_errhandler(function teams employed on fr	10
		MPI_Comm_erfnandier_ MPI_Errhandler *errh	<pre>function *comm_errhandler_fn, andler)</pre>	12
T				13
	tran 2008 l	0	rrhandler_fn, errhandler, ierror)	14
FIF I			_function), INTENT(IN) ::	15
	1100220112	comm_errhandler_fn		16 17
	TYPE(MPI_H	Errhandler), INTENT(DUT) :: errhandler	18
	INTEGER, (OPTIONAL, INTENT(OUT)) :: ierror	19
For	tran bindir	ng	· · · · ·	20
MPI_	COMM_CREAT	TE_ERRHANDLER(COMM_E	RRHANDLER_FN, ERRHANDLER, IERROR)	21
		COMM_ERRHANDLER_FN		22 23
	INTEGER EF	RRHANDLER, IERROR		23 24
	Creates an	error handler that can b	e attached to communicators.	25
		utine should be, in C, a f	$nction \ of \ type \ MPI_Comm_errhandler_function, \ w$	hich 26
	efined as		function (NDT Communication and	27
type	aei voia r	<pre>MP1_Comm_errnandler_:);</pre>	<pre>function(MPI_Comm *comm, int *error_cod</pre>	
	T			29
notu		-	icator in use. The second is the error code to ed the error. If the routine would have return	o be
			e returned in the status for the request that can	20
			naining arguments are "varargs" arguments w	
			dependent. An implementation should clearly	doc- ³⁴
			sed so that the handler may be written in Fort	26
		n mpi_f08 module, the	user routine comm_errhandler_fn should be of	the 37
form	I: FRACT INTER	SFACE		38
			function(comm, error_code)	39
		Comm) :: comm	· · · · ·	40
	INTEGER :	: error_code		41 42
Wit	h the Fortra	an mpi module and mpi	f.h, the user routine COMM_ERRHANDLER	
	ild be of the		,	44
SUBF		MM_ERRHANDLER_FUNCTI	DN(COMM, ERROR_CODE)	45
	INTEGER CO	OMM, ERROR_CODE		46
				47

	418		CHAPTER	9. MPI ENVIE	RONMENTAL MANAGEMENT
1 2 3 4	sta		viding addition	nal information t	ed because it provides an ISO- o the error handler; without this of rationale.)
5 6 7 8 9 10	is a a " con	associated with the global" error hand	"parent" cor ler for all com	nmunicator. In municators by a	inherits the error handler that particular, the user can specify associating this handler with the r initialization. (<i>End of advice to</i>
11					
12 13	MPI_CO	MM_SET_ERRHAN	DLER(comm,	errhandler)	
14	INOUT	comm		communicator (ha	undle)
15	IN	errhandler		new error handler	for communicator (handle)
16 17 18 19	C bindi	0	dler(MPI_Com	m comm, MPI_Er	rrhandler errhandler)
20 21 22 23 24 25 26 27 28 29 30 31 32	<pre>Fortran 2008 binding MPI_Comm_set_errhandler(comm, errhandler, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Errhandler), INTENT(IN) :: errhandler INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR) INTEGER COMM, ERRHANDLER, IERROR Attaches a new error handler to a communicator. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_COMM_CREATE_ERRHANDLER.</pre>				
33 34	MPI_CO	MM_GET_ERRHAN	IDLER(comm,	errhandler)	
35 36 37 38	IN OUT	comm errhandler		communicator (ha error handler curr communicator (ha	ently associated with
39 40 41	C bindi	S	dler(MPI_Com	m comm, MPI_E1	rrhandler *errhandler)
42 43 44 45 46 47 48	MPI_Comr TYPI TYPI INTI	2008 binding n_get_errhandler E(MPI_Comm), INT E(MPI_Errhandler EGER, OPTIONAL, I binding	ENT(IN) :: c), INTENT(OU	omm T) :: errhand]	

MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR) INTEGER COMM, ERRHANDLER, IERROR				
INTEGER COMM, ERMIANDLER, TEMIOR				
		ly associated with a communicator.	4	
For exam	aple, a library function may	register at its entry point the current error handler	5	
for a commu	nicator, set its own private	e error handler for this communicator, and restore	6	
before exiting	g the previous error handles	r.	7	
			8	
9.3.2 Error	Handlers for Windows		9	
			10	
			11	
MPI_WIN_CF	REATE_ERRHANDLER(win	_errhandler_fn, errhandler)	12	
IN v	vin_errhandler_fn	user defined error handling procedure (function)	13 14	
OUT e	errhandler	MPI error handler (handle)	14 15	
			16	
C binding			10	
	_create_errhandler(18	
IIIC III I_WIII		unction *win_errhandler_fn,	19	
	MPI_Errhandler *errh			
	In I_LIIndhdiel weiin	andler)	20 21	
Fortran 200			21	
		handler_fn, errhandler, ierror)	22	
PROCEDU	RE(MPI_Win_errhandler_:	<pre>function), INTENT(IN) :: win_errhandler_fn</pre>	23	
TYPE(MP	PI_Errhandler), INTENT(OUT) :: errhandler	24 25	
INTEGER	, OPTIONAL, INTENT(OUT) :: ierror	25 26	
Fortran bin	ding		20	
	J		21	
MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR) EXTERNAL WIN_ERRHANDLER_FN			20	
INTEGER ERRHANDLER, IERROR			30	
INTEGEN	, Entimendellit, TEntion		31	
Creates	an error handler that can	be attached to a window object. The user routine	32	
should be, in	C, a function of type MPI_{-}	Win_errhandler_function which is defined as	33	
typedef voi	d MPI_Win_errhandler_f	unction(MPI_Win *win, int *error_code,	34	
);			
The first	annum ant is the mindaw is	a use the second is the energy and to be noturned	35	
		n use, the second is the error code to be returned.	36	
		ser routine win_errhandler_fn should be of the form:	37 38	
	ABSTRACT INTERFACE			
	SUBROUTINE MPI_Win_errhandler_function(win, error_code)			
	PI_Win) :: win		40	
INIEGER	a :: error_code		41	
With the For	tran mpi module and mpif.	h, the user routine WIN_ERRHANDLER_FN should	42	
be of the form			43	
	WIN_ERRHANDLER_FUNCTIO	N(WIN, ERROR_CODE)	44	
	WIN, ERROR_CODE		45	
	. –		46	
			47	
			48	

```
1
     MPI_WIN_SET_ERRHANDLER(win, errhandler)
\mathbf{2}
       INOUT
                                            window object (handle)
                win
3
       IN
                errhandler
                                            new error handler for window (handle)
4
5
6
     C binding
7
     int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
8
     Fortran 2008 binding
9
     MPI_Win_set_errhandler(win, errhandler, ierror)
10
          TYPE(MPI_Win), INTENT(IN) :: win
11
          TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
16
          INTEGER WIN, ERRHANDLER, IERROR
17
          Attaches a new error handler to a window. The error handler must be either a pre-
18
     defined error handler, or an error handler created by a call to
19
     MPI_WIN_CREATE_ERRHANDLER.
20
21
22
     MPI_WIN_GET_ERRHANDLER(win, errhandler)
23
       IN
                                             window object (handle)
                 win
^{24}
       OUT
                errhandler
                                             error handler currently associated with window
25
26
                                             (handle)
27
28
     C binding
29
     int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
30
     Fortran 2008 binding
^{31}
     MPI_Win_get_errhandler(win, errhandler, ierror)
32
          TYPE(MPI_Win), INTENT(IN) :: win
33
          TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
34
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     Fortran binding
37
     MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
38
          INTEGER WIN, ERRHANDLER, IERROR
39
         Retrieves the error handler currently associated with a window.
40
41
42
43
44
45
46
47
48
```

0.0. LIU		141	
9.3.3 E	rror Handlers for Files		1
			2
			3
MPI_FILE	_CREATE_ERRHAND	DLER(file_errhandler_fn, errhandler)	4
IN	file_errhandler_fn	user defined error handling procedure (function)	5
	errhandler		6 7
OUT	errnandler	MPI error handler (handle)	8
C hindi	2.07		9
C bindin	ig File_create_errham	dlor(10
1110 III 1_		handler_function *file_errhandler_fn,	11
		er *errhandler)	12
D auturan			13
	2008 binding	(file_errhandler_fn, errhandler, ierror)	14
		Thandler_function), INTENT(IN) ::	15
1100	file_errhand		16 17
TYPE		INTENT(OUT) :: errhandler	18
INTE	GER, OPTIONAL, INT	ENT(OUT) :: ierror	19
Fortran	binding		20
		(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)	21
	CRNAL FILE_ERRHANDL		22
	GER ERRHANDLER, IE		23
Cros	tos en orror handler tl	nat can be attached to a file object. The user routine should	24
		PI_File_errhandler_function, which is defined as	25
		andler_function(MPI_File *file, int *error_code,	26 27
-JF);		21
(T)			29
		ile in use, the second is the error code to be returned. ule, the user routine file_errhandler_fn should be of the form:	30
	INTERFACE	the user routine me_ermandler_in should be of the form.	31
		andler_function(file, error_code)	32
	C(MPI_File) :: file		33
INTE	GER :: error_code		34
With the	Fortran mpi module a	nd mpif.h, the user routine FILE_ERRHANDLER_FN should	35 36
be of the	-		37
		L_FUNCTION(FILE, ERROR_CODE)	38
INTE	GER FILE, ERROR_CO	DE	39
			40
			41
MPI_FILE	E_SET_ERRHANDLER	t(file, errhandler)	42
INOUT	file	file (handle)	43
IN	errhandler	new error handler for file (handle)	44
			45 46
C bindi	າອ		40 47
	•	r(MPI_File file, MPI_Errhandler errhandler)	48

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_File_set_errhandler(file, errhandler, ierror)
3
          TYPE(MPI_File), INTENT(IN) :: file
4
          TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
5
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     Fortran binding
7
     MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
8
          INTEGER FILE, ERRHANDLER, IERROR
9
10
         Attaches a new error handler to a file. The error handler must be either a predefined
11
     error handler, or an error handler created by a call to MPI_FILE_CREATE_ERRHANDLER.
12
13
     MPI_FILE_GET_ERRHANDLER(file, errhandler)
14
15
       IN
                 file
                                            file (handle)
16
       OUT
                errhandler
                                            error handler currently associated with file (handle)
17
18
     C binding
19
     int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
20
21
     Fortran 2008 binding
22
     MPI_File_get_errhandler(file, errhandler, ierror)
23
          TYPE(MPI_File), INTENT(IN) :: file
^{24}
          TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
25
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     Fortran binding
27
     MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
28
          INTEGER FILE, ERRHANDLER, IERROR
29
30
         Retrieves the error handler currently associated with a file.
^{31}
32
     9.3.4 Error Handlers for Sessions
33
34
35
     MPI_SESSION_CREATE_ERRHANDLER(session_errhandler_fn, errhandler)
36
37
       IN
                 session_errhandler_fn
                                            user defined error handling procedure (function)
38
                                            MPI error handler (handle)
       OUT
                 errhandler
39
40
     C binding
41
     int MPI_Session_create_errhandler(
42
                    MPI_Session_errhandler_function *session_errhandler_fn,
43
                    MPI_Errhandler *errhandler)
44
45
     Fortran 2008 binding
46
     MPI_Session_create_errhandler(session_errhandler_fn, errhandler, ierror)
47
          PROCEDURE(MPI_Session_errhandler_function), INTENT(IN) ::
48
                     session_errhandler_fn
```

TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	3
Fortran binding	4
MPI_SESSION_CREATE_ERRHANDLER(SESSION_ERRHANDLER_FN, ERRHANDLER, IERROR)	5
EXTERNAL SESSION_ERRHANDLER_FN	6
INTEGER ERRHANDLER, IERROR	7
Creates an error handler that can be attached to a session object. In C, the	8
session_errhandler_fn argument should be a function of type MPI_Session_errhandler_function,	9
which is defined as	10
typedef void MPI_Session_errhandler_function(MPI_Session *session,	11
<pre>int *error_code,);</pre>	12
The first comment is the section in one the second is the second state he actumed	13
The first argument is the session in use, the second is the error code to be returned. With the Fortron main 508 module, the cossion errhandler in argument should be of the	14
With the Fortran mpi_f08 module, the session_errhandler_fn argument should be of the form:	15
ABSTRACT INTERFACE	16
SUBROUTINE MPI_Session_errhandler_function(session, error_code)	17
TYPE(MPI_Session) :: session	18
INTEGER :: error_code	19 20
	20 21
With the Fortran mpi module and mpif.h, the SESSION_ERRHANDLER_FN argument	21
should be of the form:	23
SUBROUTINE SESSION_ERRHANDLER_FUNCTION(SESSION, ERROR_CODE) INTEGER SESSION, ERROR_CODE	24
INTEGER SESSION, ERROR_CODE	
	25
	25 26
MPL SESSION SET ERRHANDLER(session errhandler)	
MPI_SESSION_SET_ERRHANDLER(session, errhandler)	26
MPI_SESSION_SET_ERRHANDLER(session, errhandler) INOUT session (handle)	26 27
	26 27 28
INOUT session (handle)	26 27 28 29 30 31
INOUT session (handle)	26 27 28 29 30 31 32
INOUTsession(handle)INerrhandlernew error handler for session (handle)	26 27 28 29 30 31 32 33
INOUT session (handle) IN errhandler new error handler for session (handle) C binding	26 27 28 29 30 31 32 33 34
<pre>INOUT session (handle) IN errhandler new error handler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35
<pre>INOUT session (handle) IN errhandler C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36
<pre>INOUT session (handle) IN errhandler new error handler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35
<pre>INOUT session (handle) IN errhandler C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37
<pre>INOUT session (handle) IN errhandler new error handler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38
<pre>INOUT session (handle) IN errhandler (handle) IN errhandler new error handler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39
<pre>INOUT session (handle) IN errhandler (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
<pre>INOUT session (handle) IN errhandler (handle) IN errhandler new error handler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
<pre>INOUT session (handle) IN errhandler (handle) IN errhandler new error handler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
<pre>INOUT session (handle) IN errhandler (handle) IN errhandler errhandler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
<pre>INOUT session (handle) IN errhandler (mode) IN errhandler errhandler (mode) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
<pre>INOUT session (handle) IN errhandler (handle) IN errhandler errhandler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45

1 MPI_SESSION_GET_ERRHANDLER(session, errhandler) 2 IN session (handle) 3 OUT errhandler error handler currently associated with session 4 (handle) 56 C binding 7 int MPI_Session_get_errhandler(MPI_Session session, 8 9 MPI_Errhandler *errhandler) 10 Fortran 2008 binding 11 MPI_Session_get_errhandler(session, errhandler, ierror) 12TYPE(MPI_Session), INTENT(IN) :: session 13 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1516Fortran binding 17 MPI_SESSION_GET_ERRHANDLER(SESSION, ERRHANDLER, IERROR) 18 INTEGER SESSION, ERRHANDLER, IERROR 19Retrieves the error handler currently associated with a session. 20219.3.5 Freeing Errorhandlers and Retrieving Error Strings 22 2324MPI_ERRHANDLER_FREE(errhandler) 2526INOUT errhandler MPI error handler (handle) 2728C binding 29 int MPI_Errhandler_free(MPI_Errhandler *errhandler) 30 31 Fortran 2008 binding MPI_Errhandler_free(errhandler, ierror) 32 33 TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler 34INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35Fortran binding 36 MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR) 37 INTEGER ERRHANDLER, IERROR 38 39 Marks the error handler associated with errhandler for deallocation and sets errhandler 40to MPI_ERRHANDLER_NULL. The error handler will be deallocated after all the objects 41 associated with it (communicator, window, or file) have been deallocated. 4243 44 4546 47 48

MPI_ERRC)R_STRING(errorcode, string, 1	resultlen)		
IN	errorcode	Error code returned by an MPI routine		
OUT	string	Text that corresponds to the errorcode		
OUT	resultlen	Length (in printable characters) of the result returned in string		
C binding int MPI_Error_string(int errorcode, char *string, int *resultlen)				
<pre>Fortran 2008 binding MPI_Error_string(errorcode, string, resultlen, ierror) INTEGER, INTENT(IN) :: errorcode CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string INTEGER, INTENT(OUT) :: resultlen INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
Fortran binding MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR) INTEGER ERRORCODE, RESULTLEN, IERROR CHARACTER*(*) STRING				
Returns the error string associated with an error code or class. The argument string must represent storage that is at least MPI_MAX_ERROR_STRING characters long. The number of characters actually written is returned in the output argument, resultlen. This function must always be thread-safe, as defined in Section 11.6. It is one of the few routines that may be called before MPI is initialized or after MPI is finalized.				
<i>Rationale.</i> The form of this function was chosen to make the Fortran and C bindings similar. A version that returns a pointer to a string has two difficulties. First, the return string must be statically allocated and different for each error message (allowing the pointers returned by successive calls to MPI_ERROR_STRING to point to the correct message). Second, in Fortran, a function declared as returning CHARACTER*(*) can not be referenced in, for example, a PRINT statement. (<i>End of rationale.</i>)				

9.4 Error Codes and Classes

The error codes returned by MPI are left entirely to the implementation (with the exception of MPI_SUCCESS). This is done to allow an implementation to provide as much information as possible in the error code (for use with MPI_ERROR_STRING).

All MPI function calls shall return MPI_SUCCESS if and only if the specification of that function has been fulfilled at the point of return. For multiple completion functions, if the function returns MPI_ERR_IN_STATUS, the error code in each status object shall be set to MPI_SUCCESS if and only if the specification of the operation represented by the corresponding MPI_Request has been fulfilled at the point of return.

When an operation raises an error, it may not satisfy its specification (for example, a synchronizing operation may not have synchronized) and the content of the output buffers, targeted memory, or output parameters is undefined. However, a valid error code shall

Unofficial Draft for Comment Only

 $45 \\ 46$

```
1
      always be set when an operation raises an error, whether in the return value, error field in
\mathbf{2}
      the status object, or element in an array of error codes.
3
          To make it possible for an application to interpret an error code, the routine
4
      MPI_ERROR_CLASS converts any error code into one of a small set of standard error codes,
\mathbf{5}
      called error classes. Valid error classes are shown in Table 9.1 and Table 9.2.
6
          The error classes are a subset of the error codes: an MPI function may return an error
7
      class number; and the function MPI_ERROR_STRING can be used to compute the error
8
      string associated with an error class. The values defined for MPI error classes are valid MPI
9
      error codes.
10
          The error codes satisfy,
11
                      0 = MPI_SUCCESS < MPI_ERR_... < MPI_ERR_LASTCODE.
12
13
           Rationale. The difference between MPI_ERR_UNKNOWN and MPI_ERR_OTHER is that
14
           MPI_ERROR_STRING can return useful information about MPI_ERR_OTHER.
15
16
           Note that MPI_SUCCESS = 0 is necessary to be consistent with C practice; the sepa-
17
           ration of error classes and error codes allows us to define the error classes this way.
18
           Having a known LASTCODE is often a nice sanity check as well. (End of rationale.)
19
20
21
      MPI_ERROR_CLASS(errorcode, errorclass)
22
23
       IN
                  errorcode
                                               Error code returned by an MPI routine
^{24}
        OUT
                  errorclass
                                               Error class associated with errorcode
25
26
      C binding
27
      int MPI_Error_class(int errorcode, int *errorclass)
28
29
     Fortran 2008 binding
30
      MPI_Error_class(errorcode, errorclass, ierror)
^{31}
          INTEGER, INTENT(IN) :: errorcode
32
          INTEGER, INTENT(OUT) :: errorclass
33
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
      Fortran binding
35
      MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
36
          INTEGER ERRORCODE, ERRORCLASS, IERROR
37
38
          The function MPI_ERROR_CLASS maps each standard error code (error class) onto
39
      itself.
40
          This function must always be thread-safe, as defined in Section 11.6. It is one of the
41
      few routines that may be called before MPI is initialized or after MPI is finalized.
42
43
      9.5
            Error Classes, Error Codes, and Error Handlers
44
45
```

Users may want to write a layered library on top of an existing MPI implementation, and this library may have its own set of error codes and classes. An example of such a library is an I/O library based on MPI, see Chapter 14. For this purpose, functions are needed to:

	No emer	1
MPI_SUCCESS MPI_ERR_ACCESS	No error Permission denied	2
MPI_ERR_AMODE	Error related to the amode passed to	3
	MPI_FILE_OPEN	4
MPI_ERR_ARG	Invalid argument of some other kind	5
MPI_ERR_ASSERT	Invalid assert argument	6
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	7
MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM	8
MPI_ERR_BUFFER	Invalid buffer pointer	9
MPI_ERR_COMM	Invalid communicator	10
MPI_ERR_CONVERSION	An error occurred in a user supplied data	11
	conversion function.	12
MPI_ERR_COUNT	Invalid count argument	13
MPI_ERR_DIMS	Invalid dimension argument	14
MPI_ERR_DISP	Invalid disp argument	15
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	16
	tered because a data representation identi-	17
	fier that was already defined was passed to	18
	MPI_REGISTER_DATAREP	19
MPI_ERR_FILE	Invalid file handle	20
MPI_ERR_FILE_EXISTS	File exists	21
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	22
	the file is currently open by some process	23
MPI_ERR_GROUP	Invalid group	24
MPI_ERR_INFO	Invalid info argument	25
MPI_ERR_INFO_KEY	Key longer than MPI_MAX_INFO_KEY	26
MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE	27
MPI_ERR_INFO_VALUE	Value longer than MPI_MAX_INFO_VAL	28
MPI_ERR_IN_STATUS	Error code is in status	29
MPI_ERR_INTERN	Internal MPI (implementation) error	30
MPI_ERR_IO	Other I/O error	31
MPI_ERR_KEYVAL	Invalid keyval has been passed	32
MPI_ERR_LOCKTYPE	Invalid locktype argument	33
MPI_ERR_NAME	Invalid service name passed to MPI_LOOKUP_NAME	34
	MPI_LOOKOF_NAME MPI_ALLOC_MEM failed because memory	35 36
MPI_ERR_NO_MEM	is exhausted	30
MPI_ERR_NO_SPACE	Not enough space	38
MPI_ERR_NO_SUCH_FILE	File does not exist	39
MPI_ERR_NOT_SAME	Collective argument not identical on all	40
	processes, or collective routines called in	41
	a different order by different processes	42
		43
		44
		45
Table 9	.1: Error classes (Part 1)	46

Unofficial Draft for Comment Only

1	MPI_ERR_OP	Invalid operation
2	MPI_ERR_OTHER	Known error not in this list
3	MPI_ERR_PENDING	Pending request
4	MPI_ERR_PORT	Invalid port name passed to
5		MPI_COMM_CONNECT
6	MPI_ERR_PROC_ABORTED	Operation failed because a peer process has
7		aborted
8	MPI_ERR_QUOTA	Quota exceeded
9	MPI_ERR_RANK	Invalid rank
10	MPI_ERR_READ_ONLY	Read-only file or file system
11	MPI_ERR_REQUEST	Invalid request (handle)
12	MPI_ERR_RMA_ATTACH	Memory cannot be attached (e.g., because
13		of resource exhaustion)
14	MPI_ERR_RMA_CONFLICT	Conflicting accesses to window
15	MPI_ERR_RMA_FLAVOR	Passed window has the wrong flavor for the
16		called function
17	MPI_ERR_RMA_RANGE	Target memory is not part of the win-
18		dow (in the case of a window created
19		with MPI_WIN_CREATE_DYNAMIC, tar-
20		get memory is not attached)
21	MPI_ERR_RMA_SHARED	Memory cannot be shared (e.g., some pro-
22		cess in the group of the specified commu-
23		nicator cannot expose shared memory)
24	MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls
25	MPI_ERR_ROOT	Invalid root
26	MPI_ERR_SERVICE	Invalid service name passed to
27		MPI_UNPUBLISH_NAME
28	MPI_ERR_SESSION	Invalid session argument
29	MPI_ERR_SIZE	Invalid size argument
30	MPI_ERR_SPAWN	Error in spawning processes
31	MPI_ERR_TAG	Invalid tag argument
32	MPI_ERR_TOPOLOGY	Invalid topology
33	MPI_ERR_TRUNCATE	Message truncated on receive
34	MPI_ERR_TYPE	Invalid datatype argument
35	MPI_ERR_UNKNOWN	Unknown error
36	MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to
37		MPI_FILE_SET_VIEW
38	MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on
39		a file which supports sequential access only
40	MPI_ERR_WIN	Invalid win argument
41	MPI_ERR_LASTCODE	Last error code
42		
43		
44	Table 9.2: Err	or classes (Part 2)
45		
46		
47		
48		

Several functions are provided to do this. They are all local. No functions are provided to free error classes or codes: it is not expected that an application will generate them in significant numbers.

MPI_AD	D_ERROR_CLASS(errorclass	s)	12
OUT	errorclass	value for the new error class (integer)	13 14
			14
C bind	0		16
int MPI	_Add_error_class(int *e	errorclass)	17
	n 2008 binding		18
	l_error_class(errorclass		19 20
	EGER, INTENT(OUT) :: er EGER, OPTIONAL, INTENT(21
			22
	n binding		23
)_ERROR_CLASS(ERRORCLASS) EGER ERRORCLASS, IERROR		24
			25
Cre	ates a new error class and r	eturns the value for it.	26 27
Ra	<i>ationale.</i> To avoid conflicts	s with existing error codes and classes, the value is set	28
		t by the user. (End of rationale.)	29
4	duise to users. Cince sell t	MDL ADD EDDOD CLASS is local, the same envertees	30
		b MPI_ADD_ERROR_CLASS is local, the same errorclass cesses that make this call. Thus, it is not safe to assume	31
		a set of processes at the same time will yield the same	32 33
		es. Getting the "same" error on multiple processes may	33 34
no	t cause the same value of er	ror code to be generated. (End of advice to users.)	35
Th	where of MDL CDD LASTCO	DE is a constant value and is not affected by new user-	36
		tead, a predefined attribute key MPI_LASTUSEDCODE is	37
		D. The attribute value corresponding to this key is the	38
		ing the user-defined ones. This is a local value and may	39
be differ	ent on different processes. T	The value returned by this key is always greater than or	40 41
equal to	MPI_ERR_LASTCODE.		41
A	<i>lvice to users</i> The value re	turned by the key MPI_LASTUSEDCODE will not change	43
		to explicitly add an error class/code. In a multithreaded	44
		ake extra care in assuming this value has not changed.	45
		cor classes are not necessarily dense. A user may not	46
as	sume that each error class b	below MPI_LASTUSEDCODE is valid. (End of advice to	47

Unofficial Draft for Comment Only

users.)

 $\mathbf{2}$

 $\overline{7}$

 48

```
1
     MPI_ADD_ERROR_CODE(errorclass, errorcode)
2
       IN
                 errorclass
                                              error class (integer)
3
       OUT
                 errorcode
                                              new error code to be associated with errorclass
4
                                              (integer)
5
6
     C binding
7
8
     int MPI_Add_error_code(int errorclass, int *errorcode)
9
     Fortran 2008 binding
10
     MPI_Add_error_code(errorclass, errorcode, ierror)
11
          INTEGER, INTENT(IN) :: errorclass
12
          INTEGER, INTENT(OUT) :: errorcode
13
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     Fortran binding
16
     MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
17
          INTEGER ERRORCLASS, ERRORCODE, IERROR
18
          Creates new error code associated with errorclass and returns its value in errorcode.
19
20
                      To avoid conflicts with existing error codes and classes, the value of the
           Rationale.
21
           new error code is set by the implementation and not by the user. (End of rationale.)
22
23
^{24}
25
     MPI_ADD_ERROR_STRING(errorcode, string)
26
       IN
                                              error code or class (integer)
                 errorcode
27
       IN
                 string
                                              text corresponding to errorcode (string)
28
29
     C binding
30
     int MPI_Add_error_string(int errorcode, const char *string)
^{31}
32
     Fortran 2008 binding
33
     MPI_Add_error_string(errorcode, string, ierror)
34
          INTEGER, INTENT(IN) :: errorcode
35
          CHARACTER(LEN=*), INTENT(IN) :: string
36
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
     Fortran binding
38
     MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
39
          INTEGER ERRORCODE, IERROR
40
41
          CHARACTER*(*) STRING
42
          Associates an error string with an error code or class. The string must be no more
43
     than MPI_MAX_ERROR_STRING characters long. The length of the string is as defined in the
44
     calling language. The length of the string does not include the null terminator in C. Trailing
45
     blanks will be stripped in Fortran. Calling MPI_ADD_ERROR_STRING for an errorcode that
46
     already has a string will replace the old string with the new string. It is erroneous to call
47
     MPI_ADD_ERROR_STRING for an error code or class with a value < MPI_ERR_LASTCODE.
```

If MPI.	_ERROR_STRING is called w	hen no string has been set, it will return a empty	1
string (all s	paces in Fortran, "" in C).		2
Section	9.3 describes the methods	for creating and associating error handlers with	3
communica	tors, files, windows, and sessi	ons.	4
			5
	/_CALL_ERRHANDLER(com	m errorcode)	6
	,	,	7 8
IN	comm	communicator with error handler (handle)	9
IN	errorcode	error code (integer)	10
			11
C binding			12
int MPI_Co	omm_call_errhandler(MPI_0	Comm comm, int errorcode)	13
Fortran 20	008 binding		14
	call_errhandler(comm, er	corcode, ierror)	15
	<pre>4PI_Comm), INTENT(IN) ::</pre>		16
	ER, INTENT(IN) :: errorco		17
INTEGH	ER, OPTIONAL, INTENT(OUT)) :: ierror	18
Fortran bi	inding		19
	CALL_ERRHANDLER(COMM, ERI		20
	ER COMM, ERRORCODE, IERR		21
This function invokes the error handler assigned to the communicator with the error			23 24
code supplied. This function returns MPI_SUCCESS in C and the same value in IERROR if			24 25
	-	(assuming the process is not aborted and the error	26
handler ret	urns).		27
			28
MPI_WIN_	CALL_ERRHANDLER(win, err	rorcode)	29
IN	win	window with error handler (handle)	30
			31
IN	errorcode	error code (integer)	32
			33
C binding			34
int MPI_WI	in_call_errhandler(MPI_W	in win, int errorcode)	35
Fortran 20	008 binding		36
MPI_Win_ca	all_errhandler(win, error	ccode, ierror)	37 38
	<pre>MPI_Win), INTENT(IN) :: </pre>		39
	ER, INTENT(IN) :: errorco		40
INTEGE	ER, OPTIONAL, INTENT(OUT)) :: ierror	41
Fortran bi	inding		42
	ALL_ERRHANDLER(WIN, ERRO	RCODE, IERROR)	43
	ER WIN, ERRORCODE, IERROF		44
			45
		adder assigned to the window with the error code	46
supplied. 1	ms function returns MPI_SU	CCESS in C and the same value in IERROR if the	47

1 2 3		handler was succes er returns).	sfully called (assuming the process is not aborted and the error
4 5 6 7		Advice to users. MPI_ERRORS_ARE_ advice to users.)	In contrast to communicators, the error handler $FATAL$ is associated with a window when it is created. (End of
8			
9	MPI_	FILE_CALL_ERRHA	NDLER(fh, errorcode)
10 11	IN	fh	file with error handler (handle)
12	IN	errorcode	error code (integer)
13			
14	C bi	nding	
15 16	int M	<pre>MPI_File_call_err</pre>	chandler(MPI_File fh, int errorcode)
17	Forti	an 2008 binding	
18			ller(fh, errorcode, ierror)
19		TYPE(MPI_File), I	
20		INTEGER, INTENT(I	.N) :: errorcode ., INTENT(OUT) :: ierror
21 22			
23		an binding	DLER(FH, ERRORCODE, IERROR)
24 25		INTEGER FH, ERROF	
26 27 28	This f	function returns MP	s the error handler assigned to the file with the error code supplied. I_SUCCESS in C and the same value in IERROR if the error handler ssuming the process is not aborted and the error handler returns).
29 30 31		Advice to users. T advice to users.)	The default error handler for files is $MPI_ERRORS_RETURN.$ (End of
32			
33			
34	MPI_	SESSION_CALL_ER	RHANDLER(session, errorcode)
35 36	IN	session	session with error handler (handle)
37	IN	errorcode	error code (integer)
38			
39		nding	
40	int N	<pre>IPI_Session_call_</pre>	errhandler(MPI_Session session, int errorcode)
41 42	Forti	an 2008 binding	
43			andler(session, errorcode, ierror)
44			, INTENT(IN) :: session
45		INTEGER, INTENT(I	N) :: errorcode ., INTENT(OUT) :: ierror
46			, INILAI(001/ ICIIOI
47 48		an binding SESSION_CALL_ERRF	HANDLER (SESSION, ERRORCODE, IERROR)

INTEGER SESSION, ERRORCODE, IERROR

This function invokes the error handler assigned to the session with the error code supplied. This function returns MPI_SUCCESS in C and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. Users are warned that handlers should not be called recursively with MPI_COMM_CALL_ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, MPI_WIN_CALL_ERRHANDLER, or MPI_SESSION_CALL_ERRHANDLER. Doing this can create a situation where an infinite recursion is created. This can occur if MPI_COMM_CALL_ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, MPI_WIN_CALL_ERRHANDLER, or MPI_SESSION_CALL_ERRHANDLER is called inside an error handler.

Error codes and classes are associated with a process. As a result, they may be used in any error handler. Error handlers should be prepared to deal with any error code they are given. Furthermore, it is good practice to only call an error handler with the appropriate error codes. For example, file errors would normally be sent to the file error handler. (End of advice to users.)

Timers and Synchronization 9.6

MPI defines a timer. A timer is specified even though it is not "message-passing," because timing parallel programs is important in "performance debugging" and because existing timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high resolution timers. See also Section 2.6.4.

MPI_WTIME()

```
C binding
double MPI_Wtime(void)
Fortran 2008 binding
DOUBLE PRECISION MPI_Wtime()
Fortran binding
DOUBLE PRECISION MPI WTIME()
    MPI_WTIME returns a floating-point number of seconds, representing elapsed wall-
clock time since some time in the past.
    The "time in the past" is guaranteed not to change during the life of the process.
The user is responsible for converting large numbers of seconds to other units if they are
preferred.
```

This function is portable (it returns seconds, not "ticks"), it allows high-resolution, and carries no unnecessary baggage. One would use it like this:

ł

```
double starttime, endtime;
starttime = MPI_Wtime();
```

2

3

4

5

6 7

8

9

10

11

12

13

14

15

16

17

18

19 2021

22

23 24

25

26

2728

29 30

31

32 33

34

35

36

37 38

39

40

41

42

43

44

4546

47

```
1
                 stuff to be timed
                                       . . .
           . . .
\mathbf{2}
                       = MPI_Wtime();
          endtime
3
          printf("That took %f seconds\n", endtime-starttime);
4
      }
5
          The times returned are local to the node that called them. There is no requirement
6
      that different nodes return "the same time." (But see also the discussion of
\overline{7}
      MPI_WTIME_IS_GLOBAL in Section 9.1.2).
8
9
10
      MPI_WTICK()
11
12
      C binding
13
      double MPI_Wtick(void)
14
15
      Fortran 2008 binding
16
      DOUBLE PRECISION MPI_Wtick()
17
      Fortran binding
18
      DOUBLE PRECISION MPI_WTICK()
19
20
          MPI_WTICK returns the resolution of MPI_WTIME in seconds. That is, it returns,
21
      as a double precision value, the number of seconds between successive clock ticks. For
22
      example, if the clock is implemented by the hardware as a counter that is incremented
23
      every millisecond, the value returned by MPI_WTICK should be (10^{-3}).
^{24}
25
26
27
28
29
30
^{31}
32
33
34
35
36
37
38
39
40
^{41}
42
43
44
45
46
47
48
```

Chapter 10

The Info Object

Many of the routines in MPI take an argument info. info is an opaque object with a handle of type MPI_Info in C and Fortran with the mpi_f08 module, and INTEGER in Fortran with the mpi module or the include file mpif.h. It stores an unordered set of (key,value) pairs (both key and value are strings). A key can have only one value. MPI reserves several keys and requires that if an implementation uses a reserved key, it must provide the specified functionality. An implementation is not required to support these keys and may support any others not reserved by MPI.

Some info hints allow the MPI library to restrict its support for certain operations in order to improve performance or resource utilization. If an application provides such an info hint, it must be compatible with any changes in the behavior of the MPI library that are allowed by the info hint.

An implementation must support info objects as caches for arbitrary (key,value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an MPI_Info must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. MPI_INFO_GET_NKEYS, MPI_INFO_GET_NTHKEY, MPI_INFO_GET_VALUELEN, MPI_INFO_GET, and MPI_INFO_GET_STRING must retain all (key,value) pairs so that layered functionality can also use the Info object.

Keys have an implementation-defined maximum length of MPI_MAX_INFO_KEY, which is at least 32 and at most 255. Values have an implementation-defined maximum length of MPI_MAX_INFO_VAL. In Fortran, leading and trailing spaces are stripped from both. Returned values will never be larger than these maximum lengths. Both key and value are case sensitive.

Rationale. Keys have a maximum length because the set of known keys will always be finite and known to the implementation and because there is no reason for keys to be complex. The small maximum size allows applications to declare keys of size MPI_MAX_INFO_KEY. The limitation on value sizes is so that an implementation is not forced to deal with arbitrarily long strings. (*End of rationale.*)

Advice to users. MPI_MAX_INFO_VAL might be very large, so it might not be wise to declare a string of that size. (*End of advice to users.*)

When info is used as an IN or INOUT argument to any MPI routine, it is parsed before that routine returns, so that it may be read, modified or freed immediately after return.

 24

 31

 $45 \\ 46$

1 When the descriptions refer to a key or value as being a boolean, an integer, or a list, $\mathbf{2}$ they mean the string representation of these types. An implementation may define its own 3 rules for how info value strings are converted to other types, but to ensure portability, every 4 implementation must support the following representations. Valid values for a boolean must 5include the strings "true" and "false" (all lowercase). For integers, valid values must include 6 string representations of decimal values of integers that are within the range of a standard $\overline{7}$ integer type in the program. (However it is possible that not every integer is a valid value 8 for a given key.) On positive numbers, + signs are optional. No space may appear between 9 a + or - sign and the leading digit of a number. For comma separated lists, the string 10 must contain valid elements separated by commas. Leading and trailing spaces are stripped 11automatically from the types of info values described above and for each element of a comma 12separated list. These rules apply to all info values of these types. Implementations are free 13 to specify a different interpretation for values of other info keys. 1415MPI_INFO_CREATE(info) 1617OUT info info object created (handle) 18 19C binding 20int MPI_Info_create(MPI_Info *info) 21Fortran 2008 binding 22 MPI_Info_create(info, ierror) 23TYPE(MPI_Info), INTENT(OUT) :: info 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2526Fortran binding 27MPI_INFO_CREATE(INFO, IERROR) 28INTEGER INFO, IERROR 29 MPI_INFO_CREATE creates a new info object. The newly created object contains no 30 key/value pairs. 31 32 33 MPI_INFO_SET(info, key, value) 34 INOUT info info object (handle) 35 36 IN key (string) key 37 value IN value (string) 38 39 C binding 40 int MPI_Info_set(MPI_Info info, const char *key, const char *value) 41 42Fortran 2008 binding 43 MPI_Info_set(info, key, value, ierror) 44 TYPE(MPI_Info), INTENT(IN) :: info 45CHARACTER(LEN=*), INTENT(IN) :: key, value 46 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 47Fortran binding 48

MPI_INFO_SET(INFO, KEY, VALUE, IERROR) INTEGER INFO, IERROR CHARACTER*(*) KEY, VALUE 1 1 2 3 4			
the same k leading and	ey was previously set. key and d trailing spaces in key and v llowed maximums, the errors) pair to info, and overrides the value if a value for d value are null-terminated strings in C. In Fortran, value are stripped. If either key or value are larger MPI_ERR_INFO_KEY or MPI_ERR_INFO_VALUE are	5 6 7 8 9
MPI INFO	_DELETE(info, key)		11
INOUT	info	info object (handle)	12
IN	key	key (string)	13 14
	ncy	kcy (suring)	15
C binding	g		16
	nfo_delete(MPI_Info info	, const char *key)	17
Fortran 2	008 binding		18
	delete(info, key, ierror)		19 20
TYPE(MPI_Info), INTENT(IN) ::	info	21
	CTER(LEN=*), INTENT(IN)	·	22
INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	23
Fortran b	binding		24
MPI_INFO_DELETE(INFO, KEY, IERROR)			25
INTEGER INFO, IERROR			26 27
CHARACTER*(*) KEY			28
MPI_INFO_DELETE deletes a (key,value) pair from info. If key is not defined in info,			29
the call rai	ses an error of class MPI_ERR	_INFO_NOKEY.	30
			31
MPI_INFO	_GET(info, key, valuelen, value	e, flag)	32
IN	info	info object (handle)	33 34
IN	key	key (string)	35
IN	valuelen	length of value arg (integer)	36
			37
OUT	value	value (string)	38
OUT	flag	true if key defined, false if not (boolean)	39 40
	_		41
C binding		onst char *key int valuelen char *value	42
<pre>int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,</pre>			43
44			44
			45 46
			40
	CTER(LEN=*), INTENT(IN)		48

```
1
          INTEGER, INTENT(IN) :: valuelen
\mathbf{2}
          CHARACTER(LEN=valuelen), INTENT(OUT) :: value
3
          LOGICAL, INTENT(OUT) :: flag
4
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     Fortran binding
6
     MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
7
          INTEGER INFO, VALUELEN, IERROR
8
          CHARACTER*(*) KEY, VALUE
9
          LOGICAL FLAG
10
11
          This function retrieves the value associated with key in a previous call to
12
     MPI_INFO_SET. If such a key exists, it sets flag to true and returns the value in value,
13
     otherwise it sets flag to false and leaves value unchanged. valuelen is the number of characters
14
     available in value. If it is less than the actual size of the value, the value is truncated. In
15
     C, valuelen should be one less than the amount of allocated space to allow for the null
16
     terminator.
17
          If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
18
19
     MPI_INFO_GET_VALUELEN(info, key, valuelen, flag)
20
21
       IN
                                              info object (handle)
                 info
22
       IN
                 key
                                              key (string)
23
                                              length of value arg (integer)
       OUT
                 valuelen
^{24}
25
       OUT
                                              true if key defined, false if not (boolean)
                 flag
26
27
     C binding
28
     int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
29
                     int *flag)
30
     Fortran 2008 binding
^{31}
     MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
32
          TYPE(MPI_Info), INTENT(IN) :: info
33
34
          CHARACTER(LEN=*), INTENT(IN) :: key
          INTEGER, INTENT(OUT) :: valuelen
35
          LOGICAL, INTENT(OUT) :: flag
36
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     Fortran binding
39
     MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
40
          INTEGER INFO, VALUELEN, IERROR
41
          CHARACTER*(*) KEY
42
          LOGICAL FLAG
43
          Retrieves the length of the value associated with key. If key is defined, valuelen is set to
44
     the length of its associated value and flag is set to true. If key is not defined, valuelen is not
45
     touched and flag is set to false. The length returned in C does not include the end-of-string
46
47
     character.
          If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
48
```

MPI_INFO	D_GET_STRING(i	nfo, key, buflen, value, flag)	1
IN	info	info object (handle)	2
IN	key	key (string)	3 4
INOUT	buflen	length of buffer (integer)	5
OUT	value	value (string)	6
OUT	flag	true if key defined, false if not (boolean)	7
001	liag	the firkey defined, lase if not (boolean)	8 9
C bindin	ıg		10
int MPI_	Info_get_string	g(MPI_Info info, const char *key, int *buflen,	11
	char *val	ue, int *flag)	12
Fortran	2008 binding		13
		fo, key, buflen, value, flag, ierror)	14 15
		TENT(IN) :: info	16
	GER, INTENT(INC	INTENT(IN) :: key	17
		INTENT(OUT) :: value	18
	CAL, INTENT(OUT		19 20
INTE	GER, OPTIONAL,	INTENT(OUT) :: ierror	21
Fortran	binding		22
		FO, KEY, BUFLEN, VALUE, FLAG, IERROR)	23
	GER INFO, BUFLE		24
	ACTER*(*) KEY, CAL FLAG	VALUE	25 26
			27
		s the value associated with key in a previous call to key exists, it sets flag to true and returns the value in value,	28
		we and leaves value unchanged. buflen on input is the size of the	29
	-	put of buflen it is the size of the buffer needed to store the value	30 31
-		into the function is less than the actual size needed to store the	32
		terminator in C), the value is truncated. On return, the value	33
		required buffer size to hold the value string. If buflen is set to C, buflen includes the required space for the null terminator. In	34
	-	ull terminated string in all cases where the buflen input value is	35
greater th			36 37
If key	is larger than M	PI_MAX_INFO_KEY, the call is erroneous.	38
Ada	<i>ice to users</i> . Th	e MPI_INFO_GET_STRING function can be used to obtain the	39
	· · · · · · · · · · · · · · · · · · ·	uffer for a value string by setting the buflen to 0. The returned	40
	-	sed to allocate memory before calling MPI_INFO_GET_STRING	41 42
agai	n to obtain the v	alue string. (End of advice to users.)	42
			44
			45

- $46 \\ 47$

```
1
     MPI_INFO_GET_NKEYS(info, nkeys)
2
       IN
                 info
                                            info object (handle)
3
       OUT
                 nkeys
                                            number of defined keys (integer)
4
5
6
     C binding
7
     int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
8
     Fortran 2008 binding
9
     MPI_Info_get_nkeys(info, nkeys, ierror)
10
          TYPE(MPI_Info), INTENT(IN) :: info
11
          INTEGER, INTENT(OUT) :: nkeys
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
16
          INTEGER INFO, NKEYS, IERROR
17
         MPI_INFO_GET_NKEYS returns the number of currently defined keys in info.
18
19
20
     MPI_INFO_GET_NTHKEY(info, n, key)
21
       IN
                 info
                                            info object (handle)
22
       IN
                                            key number (integer)
23
                 n
^{24}
       OUT
                 key
                                            key (string)
25
26
     C binding
27
     int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
28
     Fortran 2008 binding
29
30
     MPI_Info_get_nthkey(info, n, key, ierror)
^{31}
          TYPE(MPI_Info), INTENT(IN) :: info
32
          INTEGER, INTENT(IN) :: n
          CHARACTER(LEN=*), INTENT(OUT) :: key
33
34
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     Fortran binding
36
     MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
37
          INTEGER INFO, N, IERROR
38
          CHARACTER*(*) KEY
39
40
         This function returns the nth defined key in info. Keys are numbered 0 \dots N-1 where
41
     N is the value returned by MPI_INFO_GET_NKEYS. All keys between 0 and N-1 are
42
     guaranteed to be defined. The number of a given key does not change as long as info is not
     modified with MPI_INFO_SET or MPI_INFO_DELETE.
43
44
45
46
47
48
```

	_DUP(info, newinfo)		1 2
IN	info	info object (handle)	3
OUT	newinfo	info object (handle)	4
			5
C bindin	g		6
int MPI_]	Info_dup(MPI_Info info, M	PI_Info *newinfo)	7
Fortran 2	2008 binding		8
	_dup(info, newinfo, ierro	r)	9 10
TYPE	(MPI_Info), INTENT(IN) ::	info	11
	(MPI_Info), INTENT(OUT) :		12
INTEC	GER, OPTIONAL, INTENT(OUT) :: ierror	13
Fortran h	binding		14
MPI_INFO_	_DUP(INFO, NEWINFO, IERRO	R)	15
INTEC	ER INFO, NEWINFO, IERROR		16
MPI_I	NFO_DUP duplicates an exis	sting info object, creating a new object, with the	17
	value) pairs and the same ord		18 19
			20
	_FREE(info)		21
	· · /		22
INOUT	info	info object (handle)	23
a 1 • 11			24
C binding	-		25
int MPI_J	Info_free(MPI_Info *info)		26
	2008 binding		27 28
	free(info, ierror)		20
	(MPI_Info), INTENT(INOUT)		30
INTEC	GER, OPTIONAL, INTENT(OUT) :: lerror	31
Fortran h	pinding		32
	FREE(INFO, IERROR)		33
INTEC	GER INFO, IERROR		34
This f	function frees info and sets it	to MPI_INFO_NULL.	35
The v	alue of an info argument is in	terpreted each time the info is passed to a routine.	36 37
Changes to	o an info after return from a r	routine do not affect that interpretation.	38
			39
MPI_INFO	_CREATE_ENV(info)		40
OUT	info	info object (handle)	41
001	into	into object (nandle)	42
C bindin	σ		43
	•	<pre>char argv[], MPI_Info *info)</pre>	44
			45 46
	2008 binding		40
MFI_INIO_	_create_env(info, ierror)		48

1	TYPE(MPI_Info), INTENT(OUT) :: info
2	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
$\frac{3}{4}$	Fortran binding
5	MPI_INFO_CREATE_ENV(INFO, IERROR)
6	INTEGER INFO, IERROR
7	This routine produces an output object info with the same construction as
8	MPI_INFO_ENV as created during MPI_INIT or MPI_INIT_THREAD when the same argu-
9	ments are used. This construction is described in Section 11.2.1; however, this function can
10	be called when not using the World Model, e.g., when using the Sessions Model. This object
11	is not a direct copy or alias of the MPI_INFO_ENV object and could contain different values
12	based on the input arguments and other sources. Multiple calls to this procedure that are
13	given the same input arguments will produce info objects consistent with the definition of
14	MPI_INFO_ENV. The version for ISO C accepts the argc and argv that are provided by the
15 16	arguments to main or 0 for argc and NULL for argv. The user is responsible for freeing the
17	info object via MPI_INFO_FREE. This procedure is local.
18	This procedure must always be thread-safe, as defined in Section 11.6. It is one of the few routines that may be called before MPI is initialized or after MPI is finalized.
19	lew fournes that may be called before with is initialized of after with is infanzed.
20	Advice to users.
21	In some circumstances (e.g., when passing 0 to argc and NULL to argv in C or in Fortran
22	where such arguments do not exist), the info object may not be populated or may be
23	populated incompletely because this procedure is local and the implementation may
24	not be able to determine the correct values. Note that this could result in different
25 26	values in the resulting info object at different MPI processes.
27	(End of advice to users.)
28	
29	
30	
31	
32	
33 34	
35	
36	
37	
38	
39	
40	
41	
42	
43 44	
44 45	
46	
47	
48	

Chapter 11

Process Initialization, Creation, and Management

11.1 Introduction

MPI is primarily concerned with communication rather than process or resource management. However, it is necessary to address these issues to some degree in order to define a useful framework for communication. This chapter presents a set of MPI interfaces that allows for several approaches to MPI initialization and process management while placing minimal restrictions on the execution environment.

One goal of MPI is to achieve *source code portability*. By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does *not* say anything about how an MPI program is started or launched from the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup or initialization procedure to be performed before the complete set of MPI routines may be called.

To this end, MPI presents two models for MPI process initialization. In the World Model, an initial set of processes is created that are related by their membership in a common MPI_COMM_WORLD (see Section 11.2) communicator. In the Sessions Model (Section 11.3), an initial set of processes is also created, but the application must explicitly manage the creation of MPI groups, and hence MPI communicators. MPI_COMM_WORLD is only valid for use as a communicator in the World Model, i.e., after a successful call to MPI_INIT_THREAD and before a call to MPI_FINALIZE. An application can employ both of these Process Models concurrently. In multi-component MPI applications, for example, a component such as a library can make use of the Sessions Model to instantiate MPI resources without impacting the rest of the application.

Both of these models also support the *Dynamic Process Model* (see Section 11.7), which provides for the creation and management of additional processes after an MPI application has been started. A major impetus for the *Dynamic Process Model* comes from the PVM [25] research effort. This work has provided a wealth of experience with process management and resource control that illustrates their benefits and potential pitfalls.

In developing the *Dynamic Process Model*, the MPI Forum decided not to address resource control because it was not able to design a portable interface that would be ap-

 24

444 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

propriate for the broad spectrum of existing and potential resource and process controllers.

message-passing applications requiring process control. These include task farms, serial

applications with parallel modules, and problems that require a run-time assessment of the

Process management functionality is included in MPI to enable its use in classes of

MPI assumes that resource control is provided externally.

1

 $\mathbf{2}$

3

4

5

6 number and type of processes that should be started. 7 The following goals are central to the design of MPI process management: 8 9 • The MPI process model must apply to the vast majority of current parallel environments. 10 11 • MPI must not take over operating system responsibilities. It should instead provide a 12clean interface between an application and system software. 13 14• MPI must guarantee communication determinism in the presense of dynamic processes, 15i.e., dynamic process management must not introduce unavoidable race conditions. 16• MPI must not contain features that compromise performance. 17 18 The Dynamic Process Model addresses these issues in two ways. First, MPI remains 19 primarily a communication library. It does not manage the parallel environment in which 20a parallel program executes, though it provides a minimal interface between an application 21and external resource and process managers. 22 Second, MPI maintains a consistent concept of a communicator, regardless of how its 23members came into existence. A communicator is never changed once created, and it is 24 always created using deterministic collective operations. 25262711.2 The World Model 282911.2.1 Starting MPI Processes 30 When using the World Model, MPI is initialized by calling either MPI_INIT or 31 MPI_INIT_THREAD. 32 33 34MPI_INIT() 35 36 C binding 37 int MPI_Init(int *argc, char ***argv) 38 Fortran 2008 binding 39 MPI_Init(ierror) 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 42Fortran binding 43 MPI_INIT(IERROR) 44 INTEGER IERROR 45

- In the World Model, an MPI program must contain exactly one call to an MPI initialization routine: MPI_INIT_OF MPI_INIT_THREAD. MPI_COMM_WORLD and
- ⁴⁸ MPI_COMM_SELF are not valid for use as communicators prior to invocation of MPI_INIT or

MPI_INIT_THREAD. Subsequent calls to either of these initialization routines are erroneous. A subset of MPI functions may be invoked before MPI initialization routines are called. See Section 11.4. MPI_INIT accepts the argc and argv that are provided by the arguments to main or NULL:

```
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    /* parse arguments */
    /* main program */
    MPI_Finalize();    /* see below */
    return 0;
}
```

The Fortran version takes only IERROR.

Conforming implementations of MPI are required to allow applications to pass NULL for both the argc and argv arguments of main in C.

Failures may disrupt the execution of the program before or during MPI initialization. A high-quality implementation shall not deadlock during MPI initialization, even in the presence of failures. Except for functions with the MPI_T_ prefix, failures in MPI operations prior to or during MPI initialization are reported by invoking the initial error handler. Users can use the "mpi_initial_errhandler" info key during the launch of MPI processes (e.g., MPI_COMM_SPAWN / MPI_COMM_SPAWN_MULTIPLE, or mpiexec) to set a non-fatal initial error handler before MPI initialization. When the initial error handler is set to MPI_ERRORS_ABORT, raising an error before or during initialization aborts the local MPI process (i.e., it is similar to calling MPI_ABORT on MPI_COMM_SELF). An implementation may not always be capable of determining, before MPI initialization, what constitutes the local MPI process, or the set of connected processes. In this case, errors before initialization, the initial error handler is associated with MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator returned by MPI_COMM_GET_PARENT (if any).

Advice to implementors. Some failures may leave MPI in an undefined state, or raise an error before the error handling capabilities are fully operational, in which cases the implementation may be incapable of providing the desired error handling behavior. Of note, in some implementations, the notion of an MPI process is not clearly established in the early stages of MPI initialization (for example, when the implementation considers threads that called MPI_INIT as independent MPI processes); in this case, before MPI is initialized, the MPI_ERRORS_ABORT error handler may abort what would have become multiple MPI processes.

When a failure occurs during MPI initialization, the implementation may decide to return MPI_SUCCESS from the MPI initialization function instead of raising an error. It is recommended that an implementation masks an initialization error only when it expects that later MPI calls will result in well-specified behavior (i.e., barring additional failures, either the outcome of any call will be correct, or the call will raise an appropriate error). For example, it may be difficult for an implementation to avoid

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

446 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

unspecified behavior when the group of MPI_COMM_WORLD does not contain the same set of MPI processes at all members of the communicator, or if the communicator returned from MPI_COMM_GET_PARENT was not initialized correctly. (*End of advice* to implementors.)

After MPI is initialized, the application can access information about the execution environment by querying the predefined info object MPI_INFO_ENV. The following keys are predefined for this object, corresponding to the arguments of MPI_COMM_SPAWN or of mpiexec:

- ¹⁰ "command" Name of program executed.
- ¹² "argv" Space separated arguments to command.
- ¹⁴ "maxprocs" Maximum number of MPI processes to start.
- ¹⁵ "mpi_initial_errhandler" Name of the initial errhandler.
- ¹⁷ "**soft**" Allowed values for number of processors.
- ¹⁸ ₁₉ "host" Hostname.

1

 $\mathbf{2}$

3

4

5

13

21

27

28

29

30 31

32

33

34 35

36

37

- ²⁰ "arch" Architecture name.
- ²² "wdir" Working directory of the MPI process.
- $^{23}_{24}$ "file" Value is the name of a file in which additional information is specified.
- "thread_level" Requested level of thread support, if requested before the program started
 execution.

Note that all values are strings. Thus, the maximum number of processes is represented by a string such as ''1024'' and the requested level is represented by a string such as ''MPI_THREAD_SINGLE''.

Advice to users. If one of the "argv" arguments contains a space, there is no way to tell from the value of the "argv" info key whether a space is part of the argument or is separating different arguments. (*End of advice to users.*)

The info object MPI_INFO_ENV need not contain a (key,value) pair for each of these predefined keys; the set of (key,value) pairs provided is implementation-dependent. Implementations may provide additional, implementation specific, (key,value) pairs.

³⁸ In cases where the MPI processes were started with MPI_COMM_SPAWN_MULTIPLE ³⁹ or, equivalently, with a startup mechanism that supports multiple process specifications, ⁴⁰ then the values stored in the info object MPI_INFO_ENV at a process are those values that ⁴¹ affect the local MPI process.

42 43

44

45

```
Example 11.1 If MPI is started with a call to
```

```
mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos
```

Then the first 5 processes will have in their MPI_INFO_ENV object the pairs (command, ocean), (maxprocs, 5), and (arch, sun). The next 10 processes will have in MPI_INFO_ENV (command, atmos), (maxprocs, 10), and (arch, rs6000)

Advice to users. The values passed in MPI_INFO_ENV are the values of the arguments passed to the mechanism that started the MPI execution — not the actual value provided. Thus, the value associated with "maxprocs" is the number of MPI processes requested; it can be larger than the actual number of processes obtained, if the soft option was used. (*End of advice to users.*)

Advice to implementors. High-quality implementations will provide a (key,value) pair for each parameter that can be passed to the command that starts an MPI program. (End of advice to implementors.)

The following function may be used to initialize MPI, and to initialize the MPI thread environment, instead of MPI_INIT.

provided))
	provided)

IN	required	desired level of thread support (integer)
OUT	provided	provided level of thread support (integer)

C binding

int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)

Fortran 2008 binding

MPI_Init_thread(required, provided, ierror)	
INTEGER, INTENT(IN) :: required	
INTEGER, INTENT(OUT) :: provided	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	

Fortran binding

MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR) INTEGER REQUIRED, PROVIDED, IERROR

Advice to users. In C, the passing of argc and argv is optional, as with MPI_INIT as discussed in Section 11.2.1. In C, null pointers may be passed in their place. (*End of advice to users.*)

This call initializes MPI in the same way that a call to MPI_INIT would. In addition, it initializes the thread environment. The argument required is used to specify the desired level of thread support. The possible values are listed in increasing order of thread support.

MPI_THREAD_SINGLE Only one thread will execute.

- MPI_THREAD_FUNNELED The process may be multithreaded, but the application must ensure that only the main thread makes MPI calls (for the definition of main thread, see MPI_IS_THREAD_MAIN on page 449).
- MPI_THREAD_SERIALIZED The process may be multithreaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are "serialized").
- **MPI_THREAD_MULTIPLE** Multiple threads may call MPI, with no restrictions.

Unofficial Draft for Comment Only

 $\overline{7}$

¹ These values are monotonic; i.e., $MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED <$

² MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE.

³ Different processes in MPI_COMM_WORLD may require different levels of thread sup ⁴ port.

The call returns in provided information about the actual level of thread support that
 will be provided by MPI. It can be one of the four values listed above.

The level(s) of thread support that can be provided by MPI_INIT_THREAD will depend on the implementation, and may depend on information provided by the user before the program started to execute (e.g., with arguments to mpiexec). If possible, the call will return provided = required. Failing this, the call will return the least supported level such that provided > required (thus providing a stronger level of support than required by the user). Finally, if the user requirement cannot be satisfied, then the call will return in provided the highest supported level.

¹⁴ A **thread compliant** MPI implementation will be able to return **provided** ¹⁵ = MPI_THREAD_MULTIPLE. Such an implementation may always return **provided** ¹⁶ = MPI_THREAD_MULTIPLE, irrespective of the value of **required**.

¹⁷ An MPI library that is not thread compliant must always return provided =

¹⁸ MPI_THREAD_SINGLE, even if MPI_INIT_THREAD is called on a multithreaded process.
 ¹⁹ The library should also return correct values for the MPI calls that can be executed before
 ²⁰ initialization, even if multiple threads have been spawned.

- *Rationale.* Such code is erroneous, but if the MPI initialization is performed by a library, the error cannot be detected until MPI_INIT_THREAD is called. The requirements in the previous paragraph ensure that the error can be properly detected. (*End of rationale.*)
- 25 26 27

28

21

22

23 24

A call to MPI_INIT has the same effect as a call to MPI_INIT_THREAD with a required = MPI_THREAD_SINGLE.

Vendors may provide (implementation dependent) means to specify the level(s) of 29 thread support available when the MPI program is started, e.g., with arguments to mpiexec. 30 This will affect the outcome of calls to MPI_INIT and MPI_INIT_THREAD. Suppose, for 31 example, that an MPI program has been started so that only MPI_THREAD_MULTIPLE is 32 available. Then MPI_INIT_THREAD will return provided = MPI_THREAD_MULTIPLE, irre-33 spective of the value of required; a call to MPI_INIT will also initialize the MPI thread support 34level to MPI_THREAD_MULTIPLE. Suppose, instead, that an MPI program has been started 35 so that all four levels of thread support are available. Then, a call to MPI_INIT_THREAD 36 will return provided = required; alternatively, a call to MPI_INIT will initialize the MPI 37 thread support level to MPI_THREAD_SINGLE. 38

39 Rationale. Various optimizations are possible when MPI code is executed single-40 threaded, or is executed on multiple threads, but not concurrently: mutual exclusion 41 code may be omitted. Furthermore, if only one thread executes, then the MPI library 42can use library functions that are not thread safe, without risking conflicts with user 43 threads. Also, the model of one communication thread, multiple computation threads 44fits many applications well, e.g., if the process code is a sequential Fortran/C program 45with MPI calls that has been parallelized by a compiler for execution on an SMP node, 46in a cluster of SMPs, then the process computation is multithreaded, but MPI calls 47 will likely execute on a single thread. 48

The design accommodates a static specification of the thread support level, for environments that require static binding of libraries, and for compatibility for current multithreaded MPI codes. (*End of rationale.*)

Advice to implementors. If provided is not MPI_THREAD_SINGLE then the MPI library should not invoke C or Fortran library calls that are not thread safe, e.g., in an environment where malloc is not thread safe, then malloc should not be used by the MPI library.

Some implementors may want to use different MPI libraries for different levels of thread support. They can do so using dynamic linking and selecting which library will be linked when MPI_INIT_THREAD is invoked. If this is not possible, then optimizations for lower levels of thread support will occur only when the level of thread support required is specified at link time.

Note that **required** need not be the same value on all processes of MPI_COMM_WORLD. (*End of advice to implementors.*)

The following function can be used to query the current level of thread support.

		19
MPI_QUERY_THREAD(provided)		20
		21
OUT provided	provided level of thread support (integer)	22
		23
C binding		24
<pre>int MPI_Query_thread(int *provided)</pre>		25
Fortran 2008 binding		26
MPI_Query_thread(provided, ierror)		27
INTEGER, INTENT(OUT) :: provid		28
INTEGER, OPTIONAL, INTENT(OUT)		29
INTEGER, OFFICIAL, INTENT(001)		30
Fortran binding		31
MPI_QUERY_THREAD(PROVIDED, IERROR)		32
INTEGER PROVIDED, IERROR		33
The call returns in provided the curr	ent level of thread support, which will be the value	34
returned in provided by MPI_INIT_THR		35
	only applicable when using the World Model to	36
0	ns using both the World Model and the Sessions	37
	read support level returned in provided by	38
MPI_INIT_THREAD.		39
		40
		41
MPI_IS_THREAD_MAIN(flag)		42
OUT flag	true if calling thread is main thread, false otherwise (logical)	43
		44
	(8)	45
C binding		46
int MPI_Is_thread_main(int *flag)		47
THE HTTTP CHICAN MAIN(THE *1148)		48

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

 $\overline{7}$

8 9

10

11

12

13

14

15

16 17

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Is_thread_main(flag, ierror)
3
          LOGICAL, INTENT(OUT) :: flag
4
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
      Fortran binding
6
     MPI_IS_THREAD_MAIN(FLAG, IERROR)
7
          LOGICAL FLAG
8
          INTEGER IERROR
9
10
          This function can be called by a thread to determine if it is the main thread (the thread
11
      that called MPI_INIT or MPI_INIT_THREAD). This function is only applicable when using
12
      the World Model to initialize MPI. In the case of applications using both the World Model
13
      and the Sessions Model, this function only returns the thread support level returned in
14
      provided by MPI_INIT_THREAD.
15
          All routines listed in this section must be supported by all MPI implementations.
16
                         MPI libraries are required to provide these calls even if they do not
17
           Rationale.
18
           support threads, so that portable code that contains invocations to these functions
19
           can link correctly. MPI_INIT continues to be supported so as to provide compatibility
           with current MPI codes. (End of rationale.)
20
21
                                It is possible to spawn threads before MPI is initialized, but
           Advice to users.
22
           MPI_COMM_WORLD and MPI_COMM_SELF cannot be used until the World Model is
23
           active, i.e. until MPI_INIT_THREAD is invoked by one thread (which, thereby, be-
24
           comes the main thread). In particular, it is possible to enter the MPI execution with
25
26
           a multithreaded process.
27
           In the World Model, the level of thread support provided is a global property of the
28
           MPI process that can be specified only once, when MPI is initialized on that process (or
29
           before). Portable third party libraries have to be written so as to accommodate any
30
           provided level of thread support. Otherwise, their usage will be restricted to specific
31
           level(s) of thread support. If such a library can run only with specific level(s) of thread
32
           support, e.g., only with MPI_THREAD_MULTIPLE, then MPI_QUERY_THREAD can be
33
           used to check whether the user initialized MPI to the correct level of thread support
34
           and. (End of advice to users.)
35
36
     11.2.2 Finalizing MPI
37
38
39
      MPI_FINALIZE()
40
41
      C binding
42
      int MPI_Finalize(void)
43
      Fortran 2008 binding
44
     MPI_Finalize(ierror)
45
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
      Fortran binding
48
```

MPI_FINALIZE(IERROR) INTEGER IERROR

This routine cleans up all MPI state associated with the World Model. If an MPI program terminates normally (i.e., not due to a call to MPI_ABORT or an unrecoverable error) then each process must call MPI_FINALIZE before it exits.

Before an MPI process invokes MPI_FINALIZE, the process must perform all MPI calls needed to complete its involvement in MPI communications associated with the World Model. It must locally complete all MPI operations that it initiated and must execute matching calls needed to complete MPI communications initiated by other processes. For example, if the process executed a nonblocking send, it must eventually call MPI_WAIT, MPI_TEST, MPI_REQUEST_FREE, or any derived function; if the process is the target of a send, then it must post the matching receive; if it is part of a group executing a collective operation, then it must have completed its participation in the operation.

The call to MPI_FINALIZE does not clean up MPI state associated with objects created using MPI_SESSION_INIT and other Sessions Model methods, nor objects created using the communicator returned by MPI_COMM_GET_PARENT. See Sections 11.3 and 11.8.

The call to MPI_FINALIZE does not free objects created by MPI calls; these objects are freed using MPI_XXX_FREE calls.

MPI_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI_COMM_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 11.10.4.

The following examples illustrate these rules.

Example 11.2 The following code is correct

Process 0	Process 1
<pre>MPI_Init();</pre>	<pre>MPI_Init();</pre>
<pre>MPI_Send(dest=1);</pre>	<pre>MPI_Recv(src=0);</pre>
<pre>MPI_Finalize();</pre>	<pre>MPI_Finalize();</pre>

Example 11.3 Without a matching receive, the program is erroneous

Process 1
<pre>MPI_Init();</pre>
<pre>MPI_Finalize();</pre>

Example 11.4 This program is correct: Process 0 calls MPI_Finalize after it has executed the MPI calls that complete the send operation. Likewise, process 1 executes the MPI call that completes the matching receive operation before it calls MPI_Finalize.

 $\mathbf{2}$

```
1
         Process 0
                                           Process 1
\mathbf{2}
         _____
                                            _____
3
         MPI_Init();
                                           MPI_Init();
4
         MPI_Isend(dest=1);
                                           MPI_Recv(src=0);
5
         MPI_Request_free();
                                           MPI_Finalize();
6
         MPI_Finalize();
                                            exit();
7
         exit();
8
9
     Example 11.5 This program is correct. The attached buffer is a resource allocated by
10
     the user, not by MPI; it is available to the user after MPI is finalized.
11
12
         Process 0
                                           Process 1
13
         _____
                                            _____
14
         MPI_Init();
                                           MPI_Init();
15
         buffer = malloc(1000000);
                                           MPI_Recv(src=0);
16
                                           MPI_Finalize();
         MPI_Buffer_attach();
17
         MPI_Send(dest=1));
                                            exit();
18
         MPI_Finalize();
19
         free(buffer);
20
         exit();
21
22
     Example 11.6 This program is correct. The cancel operation must succeed, since the
23
     send cannot complete normally. The wait operation, after the call to MPI_Cancel, is local
24
     — no matching MPI call is required on process 1. Cancelling a send request by calling
25
     MPI_CANCEL is deprecated.
26
27
         Process 0
                                           Process 1
28
         _____
29
         MPI_Issend(dest=1);
                                           MPI_Finalize();
30
         MPI_Cancel();
31
         MPI_Wait();
32
         MPI_Finalize();
33
34
           Advice to implementors. Even though a process has executed all MPI calls needed to
35
           complete the communications it is involved with, such communication may not yet be
36
           completed from the viewpoint of the underlying MPI system. For example, a blocking
37
           send may have returned, even though the data is still buffered at the sender in an MPI
38
           buffer; an MPI process may receive a cancel request for a message it has completed
39
           receiving. The MPI implementation must ensure that a process has completed any
40
           involvement in MPI communication before MPI_FINALIZE returns. Thus, if a process
41
           exits after the call to MPI_FINALIZE, this will not cause an ongoing communication
42
           to fail. The MPI implementation should also complete freeing all objects marked for
43
           deletion by MPI calls that freed them. (End of advice to implementors.)
44
45
          Failures may disrupt MPI operations during and after MPI finalization. A high quality
46
     implementation shall not deadlock in MPI finalization, even in the presence of failures. The
47
     normal rules for MPI error handling continue to apply. After MPI_COMM_SELF has been
```

"freed" (see Section 11.2.4), errors that are not associated with a communicator, window, or file raise the initial error handler (set during the launch operation, see 11.8.4).

Although it is not required that all processes return from MPI_FINALIZE, it is required that, when it has not failed or aborted, at least the MPI process that was assigned rank 0 in MPI_COMM_WORLD returns, so that users can know that the MPI portion of the computation is over. In addition, in a POSIX environment, users may desire to supply an exit code for each process that returns from MPI_FINALIZE.

Note that a failure may terminate the MPI process that was assigned rank 0 in MPI_COMM_WORLD, in which case it is possible that no MPI process returns from MPI_FINALIZE.

Advice to users. Applications that handle errors are encouraged to implement all rank-specific code before the call to MPI_FINALIZE. In Example 11.7 below, the process with rank 0 in MPI_COMM_WORLD may have been terminated before, during, or after the call to MPI_FINALIZE, possibly leading to the code after MPI_FINALIZE never being executed. (*End of advice to users.*)

Example 11.7 The following illustrates the use of requiring that at least one process return and that it be known that process 0 is one of the processes that return. One wants code like the following to work no matter how many processes return.

```
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
...
MPI_Finalize();
if (myrank == 0) {
    resultfile = fopen("outfile", "w");
    dump_results(resultfile);
    fclose(resultfile);
}
exit(0);
```

11.2.3 Determining Whether MPI Has Been Initialized When Using the World Model

One of the goals of MPI is to allow for layered libraries. For a library using the World Model, it needs to know if MPI has been initialized using MPI_INIT or MPI_INIT_THREAD. In MPI the function MPI_INITIALIZED is provided to tell if MPI had been initialized using the World Model. In the World Model, once MPI has been finalized it cannot be restarted. A library needs to be able to determine this to act accordingly. To achieve this the function MPI_FINALIZED is needed.

MPI_FINALIZED is needed. MPI_INITIALIZED(flag) OUT flag Flag is true if MPI_INIT has been called and false otherwise

C binding int MPI_Initialized(int *flag) 24

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Initialized(flag, ierror)
3
          LOGICAL, INTENT(OUT) :: flag
4
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     Fortran binding
6
     MPI_INITIALIZED(FLAG, IERROR)
7
          LOGICAL FLAG
8
          INTEGER IERROR
9
10
          This routine may be used to determine whether MPI_INIT or MPI_INIT_THREAD has
11
     been called. MPI_INITIALIZED returns true if the calling process has called either of these
12
     MPI procedures. Whether MPI_FINALIZE has been called does not affect the behavior of
13
     MPI_INITIALIZED. This function must always be thread-safe, as defined in Section 11.6.
14
     This function returns false for applications using the Sessions Model exclusively.
15
16
     MPI_FINALIZED(flag)
17
18
                                              true if MPI was finalized (logical)
       OUT
                 flag
19
20
     C binding
21
     int MPI_Finalized(int *flag)
22
     Fortran 2008 binding
23
     MPI_Finalized(flag, ierror)
^{24}
          LOGICAL, INTENT(OUT) :: flag
25
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
     Fortran binding
28
     MPI_FINALIZED(FLAG, IERROR)
29
          LOGICAL FLAG
30
          INTEGER IERROR
31
          This routine returns true if MPI_FINALIZE has completed. It is valid to call
32
     MPI_FINALIZED before MPI_INIT and after MPI_FINALIZE. This function must always be
33
34
     thread-safe, as defined in Section 11.6.
35
36
              Allowing User Functions at MPI Finalization
     11.2.4
37
     In the context of the World Model, there are times in which it would be convenient to
38
     have actions happen when an MPI process finalizes MPI. For example, a routine may do
39
     initializations that are useful until the MPI job (or that part of the job that is being termi-
40
     nated in the case of dynamically created processes) finalizes MPI. This can be accomplished
41
     in MPI by attaching an attribute to MPI_COMM_SELF with a callback function. When
42
     MPI_FINALIZE is called, it will first execute the equivalent of an MPI_COMM_FREE on
43
     MPI_COMM_SELF. This will cause the delete callback function to be executed on all keys as-
44
     sociated with MPI_COMM_SELF, in the reverse order that they were set on MPI_COMM_SELF.
45
     If no key has been attached to MPI_COMM_SELF, then no callback is invoked. The "freeing"
46
     of MPI_COMM_SELF occurs before any other parts of MPI are affected. Thus, for example,
47
48
```

calling MPI_FINALIZED will return false in any of these callback functions. Once done with MPI_COMM_SELF, the order and rest of the actions taken by MPI_FINALIZE is not specified.

Advice to implementors. Since attributes can be added from any supported language, the MPI implementation needs to remember the creating language so the correct callback is made. Implementations that use the attribute delete callback on MPI_COMM_SELF internally should register their internal callbacks before returning from MPI_INIT / MPI_INIT_THREAD, so that libraries or applications will not have portions of the MPI implementation shut down before the application-level callbacks are made. (*End of advice to implementors.*)

11.3 The Sessions Model

There are a number of limitations with the World Model described in the preceding section. Among these are the following: MPI cannot be initialized from different application components without *a priori* knowledge or coordination; MPI cannot be initialized more than once; and MPI cannot be reinitialized after MPI_FINALIZE has been called. This section describes an alternative approach to MPI initialization — the Sessions Model. With this approach, an MPI application, or components of the application, can instantiate MPI resources for the specific communication needs of this component. MPI_COMM_WORLD is not valid for use as a communicator. MPI_INFO_ENV is not valid for use as an info object when only using the Sessions Model. As described in Section 11.2.1, MPI must be initialized using the World Model to use this info object.

In the Sessions Model, MPI resources can be allocated and freed multiple times in an MPI process.

As shown in Figure 11.1, when using the Sessions Model, an MPI process instantiates an *MPI Session handle*, which can be used to query the runtime system about characteristics of the job within which the process is running, as well as other system resources. Using this information, the MPI process can then create an MPI Group based on application requirements and available resources, which in turn can be used to create an MPI Communicator, Window, or File. By judicious creation of communicators, an application only needs to allocate MPI resources based on its communication requirements. Although there are existing MPI interfaces for creating communicators which can, in principle, allow for resource optimizations within an MPI implementation, this can only be done following initialization of MPI.

For multithreaded applications the Sessions Model provides fine-grain control of the thread support level for MPI objects. It is possible to specify different thread support levels when creating different *MPI Session handles*. Thus different components of an application can use different thread support levels.

The Sessions Model introduces a concept of isolation. MPI objects derived from different *MPI Session handles* shall not be intermixed with each other in a single MPI procedure call. MPI objects derived from the Sessions Model shall not be intermixed in a single MPI procedure call with MPI objects derived from the World Model. MPI objects derived from the Sessions Model shall not be intermixed in a single MPI procedure call with MPI objects derived from the communicator obtained from a call to MPI_COMM_GET_PARENT or MPI_COMM_JOIN.

This restriction does not apply to generalized requests (Section 13.2) as such requests are not associated directly with communicators or other MPI objects. Note however, the

Unofficial Draft for Comment Only

1

2

3

4

5

6

 $\overline{7}$

8

9

10 11

12 13

14 15

16

17

18

19

20

21

22 23

24

25

26

27

28 29

30

 31

32

33

34

35

36 37

38 39

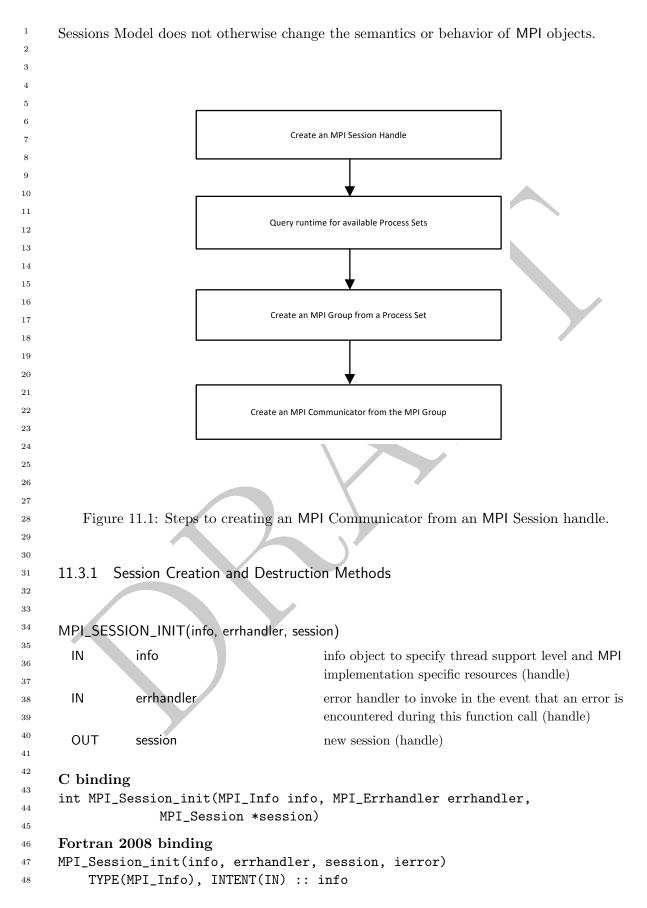
40 41

42

43 44

 $45 \\ 46$

47



TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	1			
TYPE(MPI_Session), INTENT(OUT) :: session	2 3			
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4			
Fortran binding	5			
MPI_SESSION_INIT(INFO, ERRHANDLER, SESSION, IERROR)				
INTEGER INFO, ERRHANDLER, SESSION, IERROR	7			
The info argument is used to request MPI functionality requirements and possible MPI	8			
implementation specific capabilities. The following info key is predefined:	9			
	10			
"mpi_thread_support_level" used to request the thread support level required for MPI objects derived from the Session. Allowed values are "MPI_THREAD_SINGLE",	11 12			
"MPI_THREAD_FUNNELED", "MPI_THREAD_SERIALIZED", and	13			
"MPI_THREAD_MULTIPLE". Note that the thread support value is specified by a string	14			
rather than the integer values supplied to $MPI_INIT_THREAD.$ The thread support	15			
level actually provided by the MPI implementation can be determined via a subse-	16			
quent call to MPI_SESSION_GET_INFO to return the info object associated with the	17			
Session. The default thread support level is MPI implementation dependent.	18			
The expendicy encounters are since heredler to involve in the event that the	19			
The errhandler argument specifies an error handler to invoke in the event that the Session instantiation call encounters an error. The error handler shall be either a pre-defined	20 21			
error handler (see 9.3) or one created using MPI_SESSION_CREATE_ERRHANDLER. Session	21 22			
instantiation is intended to be a lightweight operation. An MPI process may instantiate	22			
multiple Sessions. MPI_SESSION_INIT is always thread safe; multiple threads within an				
application may invoke it concurrently.	24 25			
	26			
Advice to users. Requesting "MPI_THREAD_SINGLE" thread support level is generally	27			
not recommended, because this will conflict with other components of an application				
requesting higher levels of thread support. (End of advice to users.)	29			
	30			
Advice to implementors. Owing to the restrictions of the MPI_THREAD_SINGLE	31			
thread support level, implementators are discouraged from making this the default	32			
thread support level for Sessions. (End of advice to implementors.)	33			
	34			
	35			
MPI_SESSION_FINALIZE(session)	36			
IN session session to be finalized (handle)	37 38			
	39			
C binding	40			
int MPI_Session_finalize(MPI_Session *session)	41			
Fourtheast 2008 his dia a	42			
Fortran 2008 binding	43			
<pre>MPI_Session_finalize(session, ierror) TYPE(MPI_Session), INTENT(INOUT) :: session</pre>				
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45			
	46			
Fortran binding	47			
MPI_SESSION_FINALIZE(SESSION, IERROR)	48			

1 2

6

7

8

9

10

11

12

13

14

15

16 17

18

19

20

21

22

INTEGER SESSION, IERROR

This routine cleans up all MPI state associated with the supplied session. Every instantiated Session must be finalized using MPI_SESSION_FINALIZE. The handle session is set to MPI_SESSION_NULL by the call.

Before an MPI process invokes MPI_SESSION_FINALIZE, the process must perform all MPI calls needed to complete its involvement in MPI communications: it must locally complete all MPI operations that it initiated and it must execute matching calls needed to complete MPI communications initiated by other processes.

The call to MPI_SESSION_FINALIZE does not free objects created by MPI calls; these objects are freed using MPI_XXX_FREE calls.

MPI_SESSION_FINALIZE is collective over all MPI processes that are connected via MPI Communicators, Windows, or Files that were created as part of the Session and still exist. If processes were spawned, accepted, or connected using MPI Communicators created as part of this session, this operation is collective over the union of all processes that have been and continue to be connected via those objects, as explained in Section 11.10.4.

Advice to implementors. An MPI implementation should be able to implement the semantics of MPI_SESSION_FINALIZE without synchronization with other MPI processes, provided an application frees all MPI windows, closes all MPI files, and uses MPI_COMM_DISCONNECT to free all MPI communicators associated with a session prior to invoking MPI_SESSION_FINALIZE on the corresponding session handle. (*End* of advice to implementors.)

23 24

25

11.3.2 Processes Sets

Process sets are the mechanism for MPI applications to query the runtime. Process sets are
 identified by process set names. Process set names have a Uniform Resource Identifier (URI)
 format. Two process set names are mandated: "mpi://WORLD" and

²⁹ "mpi://SELF". Additional process set names may be defined, for example,

"mpix://UNIVERSE" and "hwloc://L3Cache" may be defined by the MPI implementation. The
 "mpi://" namespace is reserved for exclusive use by the MPI standard. Figure 11.2 depicts
 process sets that the runtime could associate with an instance of an MPI job. In this
 example, the two mandated process sets are defined, in addition to optional, implementation
 specific ones.

³⁵ Mechanisms for defining process sets and how system resources are assigned to these ³⁶ sets is considered to be implementation dependent.

A process set caches key/value tuples that are accessible to the application via an MPI_Info object. The "mpi_size" key is mandatory for all process sets.

40 41

38

39

42

- 45
- 46
- 47
- 48

 23

 24

 28

 29

 41

 $\mathbf{2}$

 $\mathbf{5}$

mpi://WORLD					
location://rack/17 location://rack/23					
app://ocean app://atmos					
mpi://SELF mpi://SELF mpi://SELF mpi://SELF					
MPI process 0 MPI process 1 MPI process 2 MPI process 3 MPI process 4					
Figure 11.2: Examples of process sets. Illustrated are the two mandated process sets "mpi://WORLD" and "mpi://SELF" - along with several optional ones that a runtime coudefine. In this example, MPI_SESSION_GET_NUM_PSETS would return five at each M process.	ıld				
11.3.3 Runtime Query Functions					
MPI_SESSION_GET_NUM_PSETS(session, info, npset_names)					
IN session session (handle)					
IN info info object (handle)					
OUT npset_names number of available process sets (non-negative integer)					
C binding int MPI_Session_get_num_psets(MPI_Session session, MPI_Info info, int *npset_names)					
<pre>Fortran 2008 binding MPI_Session_get_num_psets(session, info, npset_names, ierror) TYPE(MPI_Session), INTENT(IN) :: session TYPE(MPI_Info), INTENT(IN) :: info INTEGER, INTENT(OUT) :: npset_names INTEGER, OPTIONAL, INTENT(OUT) :: ierror </pre>					
Fortran binding MPI SESSION GET NUM PSETS(SESSION, INFO, NPSET NAMES, IERROR)					

job://12942

MPI_SESSION_GET_NUM_PSETS(SESSION, INFO, NPSET_NAMES, IERROR)

460 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1 INTEGER SESSION, INFO, NPSET_NAMES, IERROR 2 This function is used to query the runtime for the number of available process sets in 3 which the calling MPI process is a member. An MPI implementation is allowed to increase 4 the number of available process sets during the execution of an MPI application when new 5process sets become available. However, MPI implementations are not allowed to change 6 the index of a particular process set name, or to change the name of the process set at a 7 particular index, or to delete a process set name once it has been added. When a process 8 set becomes invalid, for example, when some processes become unreachable due to failures 9 in the communication system, subsequent usage of the process set name should raise an 10 error. For example, creating an MPI_Group from such a process set might succeed because it 11 is a local operation, but creating an MPI_Comm from that group and attempting collective 12communication should raise an error. 13 14Advice to implementation. It is anticipated that an MPI implementation may be re-15lying on an external runtime system to provide process sets. Such runtime systems 16may have the ability to dynamically create process sets during the course of appli-17 cation execution. Requiring the number of process sets returned by 18 MPI_SESSION_GET_NUM_PSETS to be constant over the course of application exe-19 cution would prevent an application from taking advantage of such capabilities. (End 20of advice to implementors.) 2122 23 24 MPI_SESSION_GET_NTH_PSET(session, info, n, pset_len, pset_name) 25IN session (handle) session 26IN info info object (handle) 2728IN index of the desired process set name (integer) n 29 INOUT pset_len length of the pset_name argument (integer) 30 OUT name of the nth process set (string) pset_name 31 32 C binding 33 34int MPI_Session_get_nth_pset(MPI_Session session, MPI_Info info, int n, int *pset_len, char *pset_name) 35 36 Fortran 2008 binding 37 MPI_Session_get_nth_pset(session, info, n, pset_len, pset_name, ierror) 38 TYPE(MPI_Session), INTENT(IN) :: session 39 TYPE(MPI_Info), INTENT(IN) :: info 40 INTEGER, INTENT(IN) :: n 41 INTEGER, INTENT(INOUT) :: pset_len 42CHARACTER(LEN=*), INTENT(OUT) :: pset_name 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44 45Fortran binding MPI_SESSION_GET_NTH_PSET(SESSION, INFO, N, PSET_LEN, PSET_NAME, IERROR) 4647INTEGER SESSION, INFO, N, PSET_LEN, IERROR 48 CHARACTER*(*) PSET_NAME

This function returns the name of the nth process set in the supplied pset_name buffer. pset_len is the size of the buffer needed to store the nth process set name. If the pset_len passed into the function is less than the actual buffer size needed for the process set name, then the string value returned in pset_name is truncated. If pset_len is set to 0, pset_name is not changed. On return, the value of pset_len will be set to the required buffer size to hold the process set name. In C, pset_len includes the required space for the null terminator. In C, this function returns a null terminated string in all cases where the pset_len input value is greater than 0.

If two MPI processes get the same process set name, then the intersection of the two process sets shall either be the empty set or identical to the union of the two process sets.

After a successful call to MPI_SESSION_GET_NTH_PSET, subsequent calls to routines that query information about the same process set name and same session handle must return the same information. An MPI implementation is not allowed to alter any of the returned process set names.

Process set names have an implementation-defined maximum length of MPI_MAX_PSET_NAME_LEN.

Advice to users. MPI_MAX_PSET_NAME_LEN might be very large, so it might not be wise to declare a string of that size. Users are encouraged to use MPI_SESSION_GET_NTH_PSET both for obtaining the length of a pset_name and the process set name. (*End of advice to users.*)

MPI_SESSION_GET_INFO(session, info_used)				
IN	session	session (handle)	26	
OUT	info_used	see explanation below (handle)	27	
		, , ,	28	
C bindin	r A		29	
int MPI_Session_get_info(MPI_Session session, MPI_Info *info_used)				
Int MP1_Session_get_init(MP1_Session session, MP1_inito *inito_used)				
Fortran 2008 binding				
MPI_Session_get_info(session, info_used, ierror)				
TYPE(MPI_Session), INTENT(IN) :: session				
TYPE(MPI_Info), INTENT(OUT) :: info_used				
INTEC	GER, OPTIONAL, INTENT(OUT)	:: ierror	36	
Frankara 1	the dimension		37	
Fortran binding				
MPI_SESSION_GET_INFO(SESSION, INFO_USED, IERROR)				
TNLEC	GER SESSION, INFO_USED, II	LKKUK	40	

MPI_SESSION_GET_INFO returns a new info object containing the hints of the MPI 41 Session associated with session. The current setting of all hints related to this MPI Session 42is returned in info_used. An MPI implementation is required to return all hints that are 43 supported by the implementation and have default values specified; any user-supplied hints 44that were not ignored by the implementation; and any additional hints that were set by 45the implementation. If no such hints exist, a handle to a newly created info object is 46returned that contains no key/value pair. The user is responsible for freeing info_used via 47MPI_INFO_FREE. 48

1 (7

461

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8

9

10 11

12

13

14 15

16

17

18

19

20

21 22 23

462 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1 MPI_SESSION_GET_PSET_INFO(session, pset_name, info) 2 IN session session (handle) 3 IN pset_name name of process set (string) 4 5OUT info info object containing information about the given 6 process set (handle) 7 8 C binding 9 int MPI_Session_get_pset_info(MPI_Session session, const char *pset_name, 10 MPI_Info *info) 11 Fortran 2008 binding 12MPI_Session_get_pset_info(session, pset_name, info, ierror) 13 TYPE(MPI_Session), INTENT(IN) :: session 14CHARACTER(LEN=*), INTENT(IN) :: pset_name 15TYPE(MPI_Info), INTENT(OUT) :: info 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17 18 Fortran binding 19MPI_SESSION_GET_PSET_INFO(SESSION, PSET_NAME, INFO, IERROR) 20INTEGER SESSION, INFO, IERROR 21CHARACTER*(*) PSET_NAME 22 This function is used to query properties of a specific process set. The returned *info* 23object can be queried with existing MPI info object query functions. One key/value pair 24 must be defined, "mpi_size". The value of the "mpi_size" key specifies the number of MPI 25processes in the process set. The user is responsible for freeing the returned MPI_Info object. 262711.3.4 Sessions Model Examples 2829This section presents several examples of how to use MPI Sessions to create MPI Groups 30 and MPI Communicators. 31 32 Example 11.8 Simple example illustrating creation of an MPI communicator using the 33 Sessions Model. 34 35 #include <stdio.h> 36 #include <stdlib.h> 37 #include <string.h> 38#include "mpi.h" 39 40static MPI_Session lib_shandle = MPI_SESSION_NULL; 41static MPI_Comm lib_comm = MPI_COMM_NULL; 4243int library_foo_init(void) 44{ 45int rc, flag; 46int ret = 0;47const char pset_name[] = "mpi://WORLD"; 48 const char mt_key[] = "mpi_thread_support_level";

```
const char mt_value[] = "MPI_THREAD_MULTIPLE";
char out_value[100];
                       /* large enough */
MPI_Group wgroup = MPI_GROUP_NULL;
MPI_Info sinfo = MPI_INFO_NULL;
MPI_Info tinfo = MPI_INFO_NULL;
MPI_Info_create(&sinfo);
MPI_Info_set(sinfo, mt_key, mt_value);
rc = MPI_Session_init(sinfo, MPI_ERRORS_RETURN,
                       &lib_shandle);
if (rc != MPI_SUCCESS) {
   ret = -1;
   goto fn_exit;
}
/*
 * check we got thread support level foo library needs
 */
rc = MPI_Session_get_info(lib_shandle, &tinfo);
if (rc != MPI_SUCCESS) {
   ret = -1;
   goto fn_exit;
}
MPI_Info_get(tinfo, mt_key, sizeof(out_value),
             out_value, &flag);
if (flag != 1) {
   printf("Could not find key %s\n", mt_key);
   ret = -1;
   goto fn_exit;
}
if (strcmp(out_value, mt_value)) {
   printf("Did not get thread multiple support, got %s\n",
          out_value);
   ret = -1;
   goto fn_exit;
}
/*
 * create a group from the WORLD process set
 */
rc = MPI_Group_from_session_pset(lib_shandle,
                                 pset_name,
                                  &wgroup);
if (rc != MPI_SUCCESS) {
   ret = -1;
   goto fn_exit;
```

1

 $\mathbf{2}$

3

4

5 6

7

9 10

11

12 13

14

15 16

17

18

19

20 21

22

23 24 25

26

27

28 29

30

31 32 33

34

35

36 37

38 39

40 41

42

43 44

45

46

47

```
1
         }
\mathbf{2}
3
         /*
4
          * get a communicator
5
          */
6
         rc = MPI_Comm_create_from_group(wgroup,
7
                                              "org.mpi-forum.mpi-v4_0.example-ex10_8",
8
                                              MPI_INFO_NULL,
9
                                              MPI_ERRORS_RETURN,
10
                                              &lib_comm);
11
         if (rc != MPI_SUCCESS) {
12
            ret = -1;
13
            goto fn_exit;
14
         }
15
16
         /*
17
          * free group, library doesn't need it.
18
          */
19
20
     fn_exit:
21
         MPI_Group_free(&wgroup);
22
23
         if (sinfo != MPI_INFO_NULL) {
^{24}
            MPI_Info_free(&sinfo);
25
         }
26
27
         if (tinfo != MPI_INFO_NULL) {
28
            MPI_Info_free(&tinfo);
29
         }
30
^{31}
         if (ret != 0) {
32
            MPI_Session_finalize(&lib_shandle);
33
         }
34
35
         return ret;
36
     }
37
         Example 11.8 shows how the pre-defined "mpi://WORLD" process set can be used to
38
```

Example 11.8 shows how the pre-defined "mpi://WORLD" process set can be used to first create a local MPI group and then subsequently to create an MPI communicator from this group.

Example 11.9 This example illustrates the use of Process Set query functions to select a Process Set to use for MPI Group creation.

```
<sup>44</sup> #include <stdio.h>
<sup>45</sup> #include <stdlib.h>
<sup>46</sup> #include <string.h>
<sup>47</sup> #include "mpi.h"
```

```
1
int main(int argc, char *argv[])
                                                                                      \mathbf{2}
{
                                                                                      3
   int i, n_psets, psetlen, rc, ret;
   int valuelen;
                                                                                      4
                                                                                      5
   int flag = 0;
                                                                                      6
   char *pset_name = NULL;
   char *info_val = NULL;
                                                                                      7
   MPI_Session shandle = MPI_SESSION_NULL;
                                                                                      9
   MPI_Info sinfo = MPI_INFO_NULL;
                                                                                      10
   MPI_Group pgroup = MPI_GROUP_NULL;
                                                                                      11
   if (argc < 2) {
                                                                                      12
      fprintf(stderr, "A process set name fragment is required\n");
                                                                                      13
                                                                                      14
      return -1;
                                                                                      15
   }
                                                                                      16
                                                                                      17
   rc = MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_RETURN, &shandle);
                                                                                      18
   if (rc != MPI_SUCCESS) {
      fprintf(stderr, "Could not initialize session, bailing out\n");
                                                                                      19
      return -1;
                                                                                      20
   }
                                                                                     21
                                                                                     22
   MPI_Session_get_num_psets(shandle, MPI_INFO_NULL, &n_psets);
                                                                                     23
                                                                                      ^{24}
   for (i=0, pset_name=NULL; i<n_psets; i++) {</pre>
                                                                                      25
                                                                                      26
       psetlen = 0;
       MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, i,
                                                                                      27
                                                                                      28
                                  &psetlen, NULL);
                                                                                      29
       pset_name = (char *)malloc(sizeof(char) * psetlen);
                                                                                      30
       MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, i,
                                                                                      31
                                  &psetlen, pset_name);
       if (strstr(pset_name, argv[1]) != NULL) break;
                                                                                      32
                                                                                      33
                                                                                      34
       free(pset_name);
       pset_name = NULL;
                                                                                      35
   }
                                                                                      36
                                                                                     37
                                                                                      38
   /*
                                                                                      39
    * get instance of an info object for this Session
                                                                                      40
    */
                                                                                      41
                                                                                      42
   MPI_Session_get_pset_info(shandle, pset_name, &sinfo);
   MPI_Info_get_valuelen(sinfo, "mpi_size", &valuelen, &flag);
                                                                                      43
                                                                                      44
   info_val = (char *)malloc(valuelen+1);
   MPI_Info_get(sinfo, "mpi_size", valuelen, info_val, &flag);
                                                                                      45
                                                                                      46
   free(info_val);
                                                                                      47
                                                                                      48
```

```
/*
```

```
1
          * create a group from the process set
\mathbf{2}
          */
3
4
        rc = MPI_Group_from_session_pset(shandle, pset_name,
5
                                              &pgroup);
6
        ret = (rc == MPI_SUCCESS) ? 0 : -1;
7
8
        free(pset_name);
9
        MPI_Group_free(&pgroup);
10
        MPI_Info_free(&sinfo);
11
        MPI_Session_finalize(&shandle);
12
13
        fprintf(stderr, "Test completed ret = %d\n", ret);
14
        return ret;
15
16
     }
17
         Example 11.9 illustrates several aspects of the Sessions Model. First, the default error
18
     handler can be specified when instantiating a Session instance. Second, there must be at
19
     least two process sets associated with a Session. Third, the example illustrates use of the
20
     Sessions info object and the one required key: "mpi_size".
21
22
     Example 11.10 A Fortran 2008 example illustrating how to obtain information about
23
     available process sets, create an MPI Group from a process set, and subsequently create an
24
     MPI Communicator.
25
26
     PROGRAM MAIN
27
         USE mpi_f08
28
         IMPLICIT NONE
29
         INTEGER :: pset_len, ierror, n_psets
30
         CHARACTER(LEN=:), ALLOCATABLE :: pset_name
31
         TYPE(MPI_Session) :: shandle
32
         TYPE(MPI_Group) :: pgroup
33
         TYPE(MPI_Comm) :: pcomm
34
35
         CALL MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_RETURN, &
36
                                 shandle, ierror)
37
         IF (ierror .NE. MPI_SUCCESS) THEN
38
             WRITE(*,*) "MPI_Session_init failed"
39
             ERROR STOP
40
         END IF
41
42
         CALL MPI_Session_get_num_psets(shandle, MPI_INFO_NULL, n_psets)
43
         IF (n_psets .LT. 2) THEN
44
             WRITE(*,*) "MPI_Session_get_num_psets didn't return at least 2 psets"
45
             ERROR STOP
46
         END IF
47
48
     ļ
```

```
1
!
    Just get the second pset's length and name
                                                                                       \mathbf{2}
!
    Note that index values are zero-based, even in Fortran
!
    pset_len = 0
                                                                                       5
    CALL MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, 1,
                                                                                       6
                                                                       &
                                     pset_len, pset_name)
    ALLOCATE(CHARACTER(LEN=pset_len)::pset_name)
                                                                                       9
    CALL MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, 1,
                                                                       &
                                                                                       10
                                     pset_len, pset_name)
                                                                                       11
ļ
                                                                                       12
                                                                                       13
!
    create a group from the pset
                                                                                       14
!
                                                                                       15
    CALL MPI_Group_from_session_pset(shandle, pset_name, pgroup)
                                                                                       16
L
                                                                                       17
i
    free the buffer used for the pset name
                                                                                       18
!
                                                                                       19
    DEALLOCATE(pset_name)
                                                                                       20
                                                                                      21
ļ
                                                                                       22
!
    create a MPI communicator from the group
                                                                                      23
ļ
                                                                                       ^{24}
                                                "session_example",
    CALL MPI_Comm_create_from_group(pgroup,
                                                                       &
                                                                                       25
                                                MPI_INFO_NULL,
                                                                       &
                                                                                       26
                                                MPI_ERRORS_RETURN,
                                                                       &
                                                pcomm)
                                                                                       27
                                                                                       28
                                                                                       29
    CALL MPI_Barrier(pcomm, ierror)
                                                                                       30
    IF (ierror .NE. MPI_SUCCESS) THEN
                                                                                       31
        WRITE(*,*) "Barrier call on communicator failed"
        ERROR STOP
                                                                                       32
    END IF
                                                                                       33
                                                                                      34
    CALL MPI_Comm_free(pcomm)
                                                                                      35
    CALL MPI_Group_free(pgroup)
                                                                                       36
                                                                                      37
    CALL MPI_Session_finalize(shandle, ierror)
                                                                                       38
                                                                                       39
END PROGRAM MAIN
                                                                                       40
                                                                                       41
    Note in this example that the call to MPI_SESSION_FINALIZE may block in order
                                                                                       42
to ensure that the calling MPI process has completed its involvement in the preceding
                                                                                       43
MPI_BARRIER operation. If MPI_COMM_DISCONNECT had been used instead of
                                                                                       44
MPI_COMM_FREE, the example would have blocked in MPI_COMM_DISCONNECT rather
                                                                                       45
than MPI_SESSION_FINALIZE.
```

```
Unofficial Draft for Comment Only
```

11.4 Common Elements of Both Process Models

11.4.1 MPI Functionality that is Always Available

1 2

3 4

5

6

 $\overline{7}$

8

33

36

37

38

Some MPI functions may be invoked at any time, including prior to calling MPI_INIT or MPI_SESSION_INIT, and following MPI finalization, independent of whether the World Model, Sessions Model, or both are used. These functions can be called concurrently by multiple threads within an MPI Process. Table 11.1 lists the applicable MPI functions.

9		MPI_INITIALIZED	
10		MPI_FINALIZED	
11		MPI_GET_VERSION	
12		MPI_GET_LIBRARY_VERSION	
13		MPI_INFO_CREATE	
14		MPI_INFO_CREATE_ENV	
15		MPI_INFO_SET	
16		MPI_INFO_DELETE	
17		MPI_INFO_GET	
18		MPI_INFO_GET_VALUELEN	
19		MPI_INFO_GET_NKEYS	
20		MPI_INFO_GET_NTHKEY	
21		MPI_INFO_DUP	
22		MPI_INFO_FREE	
23		MPI_INFO_F2C	
24		MPI_INFO_C2F	
25		MPI_SESSION_CREATE_ERRHANDLER	
26		MPI_SESSION_CALL_ERRHANDLER	
27		MPI_ERRHANDLER_FREE	
28		MPI_ERRHANDLER_F2C	
29		MPI_ERRHANDLER_C2F	
30	, i i i i i i i i i i i i i i i i i i i	MPI_ERROR_STRING	
31			
32		MPI_ERROR_CLASS	

Table 11.1: List of MPI Functions that can be called at any time within an MPI program,
 including prior to MPI initialization and following MPI finalization

In addition to the functions listed in Table 11.1, any function with the prefix MPI_T_ (within the constraints for functions with this prefix listed in Section 15.3.4) may also be called prior to MPI initialization and after MPI finalization.

MPI_ABORT(comm, errorcode)

IN	comm	communicator of tasks to abort
IN	errorcode	error code to return to invoking environment

C binding

int MPI_Abort(MPI_Comm comm, int errorcode)

Fortran 2008 binding

MPI_Abort(comm, errorcode, ierror)
 TYPE(MPI_Comm), INTENT(IN) :: comm
 INTEGER, INTENT(IN) :: errorcode
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

MPI_ABORT(COMM, ERRORCODE, IERROR) INTEGER COMM, ERRORCODE, IERROR

This routine makes a "best attempt" to abort all MPI processes in the group of comm. This function does not require that the invoking environment take any action with the error code. However, a Unix or POSIX environment should handle this as a return errorcode from the main program.

It may not be possible for an MPI implementation to abort only the processes represented by comm if this is a subset of the processes. In this case, the MPI implementation should attempt to abort all the connected processes but should not abort any unconnected processes. When using the World Model, and if no processes were spawned, accepted, or connected then this has the effect of aborting all the processes associated with MPI_COMM_WORLD. In the case of the Sessions Model, if an MPI process has instantiated multiple sessions, the union of the process sets in these sessions are considered connected processes. Thus invoking MPI_ABORT on a communicator derived from one of these sessions will result in all MPI processes in this union being aborted.

Advice to implementors. After aborting a subset of processes, a high quality implementation should be able to provide error handling for communicators, windows, and files involving both aborted and non-aborted processes. As an example, if the user changes the error handler for MPI_COMM_WORLD to MPI_ERRORS_RETURN or a custom error handler, when a subset of MPI_COMM_WORLD is aborted, the remaining processes in MPI_COMM_WORLD should be able to continue communicating with each other and receive an appropriate error code when attempting communication with an aborted process (e.g., an error of class MPI_ERR_PROC_ABORTED). A high quality implementation should support equivalent behavior for communicators derived from sessions. (End of advice to implementors.)

Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the MPI library but not mandatory. (*End of advice to users.*)

Unofficial Draft for Comment Only

 24

 31

Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. mpiexec or singleton init). (End of advice to implementors.)

11.5 Portable MPI Process Startup

A number of implementations of MPI provide a startup command for MPI programs that is of the form

9 10 11

27 28

29

34

35

36

37

38

1

2

3

4 5

6 7

8

mpirun <mpirun arguments> <program> <program arguments>

Separating the command to start the program from the program itself provides flexibility,
 particularly for network and heterogeneous implementations. For example, the startup
 script need not run on one of the machines that will be executing the MPI program itself.

Having a standard startup mechanism also extends the portability of MPI programs one step further, to the command lines and scripts that manage them. For example, a validation suite script that runs hundreds of programs can be a portable script if it is written using such a standard startup mechanism. In order that the "standard" command not be confused with existing practice, which is not standard and not portable among implementations, instead of mpirun MPI specifies mpiexec.

While a standardized startup mechanism improves the usability of MPI, the range of environments is so diverse (e.g., there may not even be a command line interface) that MPI cannot mandate such a mechanism. Instead, MPI specifies an **mpiexec** startup command and recommends but does not require it, as advice to implementors. However, if an implementation does provide a command called **mpiexec**, it must be of the form described below.

- It is suggested that
- mpiexec -n <numprocs> <program>

be at least one way to start <program> with an initial set of <numprocs> processes, which
 will be accessible as the process set named "mpi://world" in the Sessions Model and/or
 used to the form the group associated with the built-in communicator, MPI_COMM_WORLD
 in the World Model. Other arguments to mpiexec may be implementation-dependent.

Advice to implementors. Implementors, if they do provide a special startup command for MPI programs, are advised to give it the following form. The syntax is chosen in order that mpiexec be able to be viewed as a command-line version of MPI_COMM_SPAWN (See Section 11.8.4).

Analogous to MPI_COMM_SPAWN, we have

mpiexec -n	<maxp< th=""><th>orocs></th></maxp<>	orocs>
-soft	<	>
-host	<	>
-arch	<	>
-wdir	<	>
-path	<	>
-file	<	>
-initial-errhandler	<	>

. . .

<command line>

for the case where a single command line for the application program and its arguments will suffice. See Section 11.8.4 for the meanings of these arguments. For the case corresponding to MPI_COMM_SPAWN_MULTIPLE there are two possible formats: Form A:

mpiexec { <above arguments> } : { ... } : { ... } : ... ; { ... }

As with MPI_COMM_SPAWN, all the arguments are optional. (Even the $-n \ge argument$ is optional; the default is implementation dependent. It might be 1, it might be taken from an environment variable, or it might be specified at compile time.) The names and meanings of the arguments are taken from the keys in the info argument to MPI_COMM_SPAWN. There may be other, implementation-dependent arguments as well.

Note that Form A, though convenient to type, prevents colons from being program arguments. Therefore an alternate, file-based form is allowed:

Form B:

mpiexec -configfile <filename>

where the lines of < filename > are of the form separated by the colons in Form A. Lines beginning with '#' are comments, and lines may be continued by terminating the partial line with '\'.

Example 11.11 Start 16 instances of myprog on the current or default machine:

mpiexec -n 16 myprog

Example 11.12 Start 10 processes on the machine called ferrari:

mpiexec -n 10 -host ferrari myprog

Example 11.13 Start three copies of the same program with different command-line arguments:

mpiexec myprog infile1 : myprog infile2 : myprog infile3

Example 11.14 Start the ocean program on five Suns and the atmos program on 10 RS/6000's:

mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos

Unofficial Draft for Comment Only

 $\mathbf{2}$

It is assumed that the implementation in this case has a method for choosing hosts of the appropriate type. Their ranks are in the order specified.

Example 11.15 Start the ocean program on five Suns and the atmos program on 10 RS/6000's (Form B):

mpiexec -configfile myfile

where myfile contains

1

2

3 4

5

6 7

8 9

10 11

12

13 14

1516

17

32

33

37

38 39

40

41

42

43 44 -n 5 -arch sun ocean

```
-n 10 -arch rs6000 atmos
```

(End of advice to implementors.)

MPI and Threads 11.6

18 This section specifies the interaction between MPI calls and threads. Although thread com-19 pliance is not required, the standard specifies how threads are to work if they are provided. 20The section lists minimal requirements for thread compliant MPI implementations and 21defines functions that can be used for initializing the thread environment. MPI may be im-22 plemented in environments where threads are not supported or perform poorly. Therefore, 23MPI implementations are not required to be thread compliant as defined in this section. 24Regardless of whether or not the MPI implementation is thread compliant, a subset of MPI 25functions must always be thread safe. A complete list of such MPI functions is given in Ta-26ble 11.1. When a thread is executing one of these routines, if another concurrently running 27thread also makes an MPI call, the outcome will be as if the calls executed in some order. 28

This section generally assumes a thread package similar to POSIX threads [44], but the 29 syntax and semantics of thread calls are not specified here — these are beyond the scope 30 of this document. 31

11.6.1 General

34In a thread-compliant implementation, an MPI process is a process that may be multi-35 threaded. Each thread can issue MPI calls; however, threads are not separately addressable: 36 a rank in a send or receive call identifies a process, not a thread. A message sent to a process can be received by any thread in this process.

This model corresponds to the POSIX model of interprocess commu-Rationale. nication: the fact that a process is multithreaded, rather than single-threaded, does not affect the external interface of this process. MPI implementations in which MPI 'processes' are POSIX threads inside a single POSIX process are not thread-compliant by this definition (indeed, their "processes" are single-threaded). (End of rationale.)

Advice to users. It is the user's responsibility to prevent races when threads within 45the same application post conflicting communication calls. The user can make sure 4647 that two threads in the same process will not issue conflicting communication calls by using distinct communicators at each thread. (End of advice to users.) 48

The two main requirements for a thread-compliant implementation are listed below.

- 1. All MPI calls are *thread-safe*, i.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved.
- 2. Blocking MPI calls will block the calling thread only, allowing another thread to execute, if available. The calling thread will be blocked until the event on which it is waiting occurs. Once the blocked communication is enabled and can proceed, then the call will complete and the thread will be marked runnable, within a finite time. A blocked thread will not prevent progress of other runnable threads on the same process, and will not prevent them from executing MPI calls.

Example 11.16 Process 0 consists of two threads. The first thread executes a blocking send call MPI_Send(buff1, count, type, 0, 0, comm), whereas the second thread executes a blocking receive call MPI_Recv(buff2, count, type, 0, 0, comm, &status), i.e., the first thread sends a message that is received by the second thread. This communication should always succeed. According to the first requirement, the execution will correspond to some interleaving of the two calls. According to the second requirement, a call can only block the calling thread and cannot prevent progress of the other thread. If the send call went ahead of the receive call, then the sending thread may block, but this will not prevent the receiving thread from executing. Thus, the receive call will occur. Once both calls occur, the communication is enabled and both calls will complete. On the other hand, a single-threaded process that posts a send, followed by a matching receive, may deadlock. The progress requirement for multithreaded implementations is stronger, as a blocked call cannot prevent progress in other threads.

Advice to implementors. MPI calls can be made thread-safe by executing only one at a time, e.g., by protecting MPI code with one process-global lock. However, blocked operations cannot hold the lock, as this would prevent progress of other threads in the process. The lock is held only for the duration of an atomic, locally-completing suboperation such as posting a send or completing a send, and is released in between. Finer locks can provide more concurrency, at the expense of higher locking overheads. Concurrency can also be achieved by having some of the MPI protocol executed by separate server threads. (*End of advice to implementors.*)

11.6.2 Clarifications

Initialization and Completion When using the World Model, the call to MPI_FINALIZE should occur on the same thread that initialized MPI. We call this thread the **main thread**. The call should occur only after all process threads have completed their MPI calls, and have no pending communications or I/O operations.

Rationale. This constraint simplifies implementation. (End of rationale.)

Threads and the Sessions Model The Sessions Model provides a finer-grain approach to controlling the interaction between MPI calls and threads. When using this model, the desired level of thread support is specified at Session initialization time. See Section 11.3. Thus it is possible for communicators and other MPI objects derived from one Session 48

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

 24

¹ to provide a different level of thread support than those created from another Session ² for which a different level of thread support was requested. Depending on the level of ³ thread support requested at Session initialization time, different threads in a MPI process ⁴ can make concurrent calls to MPI when using MPI objects derived from different *session* ⁵ *handles.* Note that the requested and provided level of thread support when creating a ⁶ Session may influence the granted level of thread support in a subsequent invocation of ⁷ MPI_SESSION_INIT. Likewise, if the application at some point calls

⁸ MPI_INIT_THREAD, the requested and granted level of thread support may influence the ⁹ granted level of thread support for subsequent calls to MPI_SESSION_INIT. Similarly, if the ¹⁰ application calls MPI_INIT_THREAD after a call to MPI_SESSION_INIT, the level of thread ¹¹ support returned from MPI_INIT_THREAD may be similarly influenced by the requested ¹² level of thread support in the prior call to MPI_SESSION_INIT.

¹³ In addition, if an MPI application is only using the Sessions Model, the provided thread ¹⁴ support level returned by MPI_QUERY_THREAD is the same as that returned prior to ¹⁵ invocation of MPI_INIT_THREAD or MPI_INIT. If the application also used the World ¹⁶ Model in some component of the application, MPI_QUERY_THREAD will return the level ¹⁷ of thread support returned by the original call to MPI_INIT_THREAD.

¹⁹ Multiple threads completing the same request. A program in which two threads block, wait-²⁰ ing on the same request, is erroneous. Similarly, the same request cannot appear in the ²¹ array of requests of two concurrent MPI_{WAIT|TEST}{ANY|SOME|ALL} calls. In MPI, a ²² request can only be completed once. Any combination of wait or test that violates this rule ²³ is erroneous.

Rationale. This restriction is consistent with the view that a multithreaded execution corresponds to an interleaving of the MPI calls. In a single threaded implementation, once a wait is posted on a request the request handle will be nullified before it is possible to post a second wait on the same handle. With threads, an MPI_WAIT{ANY|SOME|ALL} may be blocked without having nullified its request(s)

- so it becomes the user's responsibility to avoid using the same request in an MPI_WAIT on another thread. This constraint also simplifies implementation, as only one thread will be blocked on any communication or I/O event. (*End of rationale.*)
- ³⁴ Probe A receive call that uses source and tag values returned by a preceding call to ³⁵ MPI_PROBE or MPI_IPROBE will receive the message matched by the probe call only ³⁶ if there was no other matching receive after the probe and before that receive. In a multi-³⁷ threaded environment, it is up to the user to enforce this condition using suitable mutual ³⁸ exclusion logic. This can be enforced by making sure that each communicator is used by ³⁹ only one thread on each process. Alternatively, MPI_MPROBE or MPI_IMPROBE can be ⁴⁰ used.
- 41

18

25

26

27

28

29

30

 31

32

33

⁴² Collective calls Matching of collective calls on a communicator, window, or file handle is
 ⁴³ done according to the order in which the calls are issued at each process. If concurrent
 ⁴⁴ threads issue such calls on the same communicator, window or file handle, it is up to the
 ⁴⁵ user to make sure the calls are correctly ordered, using interthread synchronization.

46

Advice to users. With three concurrent threads in each MPI process of a communica tor comm, it is allowed that thread A in each MPI process calls a collective operation

on comm, thread B calls a file operation on an existing filehandle that was formerly opened on comm, and thread C invokes one-sided operations on an existing window handle that was also formerly created on comm. (*End of advice to users.*)

Rationale. As specified in MPI_FILE_OPEN and MPI_WIN_CREATE, a file handle and a window handle inherit only the group of processes of the underlying communicator, but not the communicator itself. Accesses to communicators, window handles and file handles cannot affect one another. (*End of rationale.*)

Advice to implementors. If the implementation of file or window operations internally uses MPI communication then a duplicated communicator may be cached on the file or window object. (*End of advice to implementors.*)

Error handlers An error handler does not necessarily execute in the context of the thread that made the error-raising MPI call; the error handler may be executed by a thread that is distinct from the thread that will return the error code.

Rationale. The MPI implementation may be multithreaded, so that part of the communication protocol may execute on a thread that is distinct from the thread that made the MPI call. The design allows the error handler to be executed on the thread where the error is raised. (*End of rationale.*)

Interaction with signals and cancellations The outcome is undefined if a thread that executes an MPI call is cancelled (by another thread), or if a thread catches a signal while executing an MPI call. However, a thread of an MPI process may terminate, and may catch signals or be cancelled by another thread when not executing MPI calls.

Rationale. Few C library functions are signal safe, and many have cancellation points — points at which the thread executing them may be cancelled. The above restriction simplifies implementation (no need for the MPI library to be "async-cancel-safe" or "async-signal-safe"). (*End of rationale.*)

Advice to users. Users can catch signals in separate, non-MPI threads (e.g., by masking signals on MPI calling threads, and unmasking them in one or more non-MPI threads). A good programming practice is to have a distinct thread blocked in a call to sigwait for each user expected signal that may occur. Users must not catch signals used by the MPI implementation; as each MPI implementation is required to document the signals used internally, users can avoid these signals. (*End of advice to users.*)

Advice to implementors. The MPI library should not invoke library calls that are not thread safe, if multiple threads execute. (*End of advice to implementors.*)

11.7 The Dynamic Process Model

The dynamic process model allows for the creation and cooperative termination of processes 44 after an MPI application has started. It provides a mechanism to establish communication 45 between the newly created processes and the existing MPI application. It also provides a 46 mechanism to establish communication between two existing MPI applications, even when 47 one did not "start" the other. 48

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

 24

11.7.1 Starting Processes

MPI applications may start new processes through an interface to an external process manager.

MPI_COMM_SPAWN starts MPI processes and establishes communication with them, returning an intercommunicator. MPI_COMM_SPAWN_MULTIPLE starts several different 6 binaries (or the same binary with different arguments), placing them in the same MPI_COMM_WORLD and returning an intercommunicator.

MPI uses the group abstraction to represent processes. A process is identified by a (group, rank) pair.

10 11 12

39 40

41

42

43

44

4546

47

48

1

 $\mathbf{2}$

3

4

5

7

8

9

11.7.2 The Runtime Environment

13 The MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE routines provide an inter-14face between MPI and the *runtime environment* of an MPI application. The difficulty is 15that there is an enormous range of runtime environments and application requirements, and 16MPI must not be tailored to any particular one.

17MPI assumes, implicitly, the existence of an environment in which an application runs. 18 It does not provide "operating system" services, such as a general ability to query what 19processes are running, to kill arbitrary processes, to find out properties of the runtime 20environment (how many processors, how much memory, etc.). Complex interaction of an 21MPI application with its runtime environment should be done through an environment-22specific API.

23At some low level, obviously, MPI must be able to interact with the runtime system, 24 but the interaction is not visible at the application level and the details of the interaction 25are not specified by the MPI standard.

26In many cases, it is impossible to keep environment-specific information out of the MPI 27interface without seriously compromising MPI functionality. To permit applications to take 28advantage of environment-specific functionality, many MPI routines take an info argument 29that allows an application to specify environment-specific information. There is a tradeoff 30 between functionality and portability: applications that make use of environment-specific 31 info are not portable.

32 MPI does not require the existence of an underlying "virtual machine" model, in which 33 there is a consistent global view of an MPI application and an implicit "operating system" 34managing resources and processes. For instance, processes spawned by one task may not 35 be visible to another; additional hosts added to the runtime environment by one process 36 may not be visible in another process; tasks spawned by different processes may not be 37 automatically distributed over available resources. 38

Interaction between MPI and the runtime environment is limited to the following areas:

• A process may start new processes with MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE.

- When a process spawns a child process, it may optionally use an info argument to tell the runtime environment where or how to start the process. This extra information may be opaque to MPI.
- An attribute MPI_UNIVERSE_SIZE (See Section 11.10.1) on MPI_COMM_WORLD tells a program how "large" the initial runtime environment is, namely how many processes

can usefully be started in all. One can subtract the size of MPI_COMM_WORLD from this value to find out how many processes might usefully be started in addition to those already running.

11.8 Process Manager Interface

11.8.1 Processes in MPI

A process is represented in MPI by a (group, rank) pair. A (group, rank) pair specifies a unique process but a process does not determine a unique (group, rank) pair, since a process may belong to several groups.

11.8.2 Starting Processes and Establishing Communication

The following routine starts a number of MPI processes and establishes communication with them, returning an intercommunicator.

Advice to users. It is possible in MPI to start an SPMD or MPMD application with a fixed number of processes after initialization by first starting one process and having that process start its siblings with MPI_COMM_SPAWN. This practice is discouraged primarily for reasons of performance. If possible, it is preferable to start all processes at once, as a single MPI application. (*End of advice to users.*)

MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, array_of_errcodes)

IN	command	name of program to be spawned (string, significant only at root)	27 28 29
IN	argv	arguments to command (array of strings, significant only at root)	30 31
IN	maxprocs	maximum number of processes to start (integer, significant only at root)	32 33
IN	info	a set of key-value pairs telling the runtime system where and how to start the processes (handle, significant only at root)	34 35 36
IN	root	rank of process in which previous arguments are examined (integer)	37 38 39
IN	comm	intracommunicator containing group of spawning processes (handle)	40 41
OUT	intercomm	intercommunicator between original group and the newly spawned group (handle)	42 43 44
OUT	array_of_errcodes	one code per process (array of integer)	44 45 46
C binding 47			

 $\frac{4}{5}$

```
1
     int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
\mathbf{2}
                    MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,
3
                    int array_of_errcodes[])
4
     Fortran 2008 binding
5
     MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
6
                    array_of_errcodes, ierror)
7
          CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
8
          INTEGER, INTENT(IN) :: maxprocs, root
9
          TYPE(MPI_Info), INTENT(IN) :: info
10
          TYPE(MPI_Comm), INTENT(IN) :: comm
11
          TYPE(MPI_Comm), INTENT(OUT) :: intercomm
12
          INTEGER :: array_of_errcodes(*)
13
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     Fortran binding
16
     MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
17
                    ARRAY_OF_ERRCODES, IERROR)
18
          CHARACTER*(*) COMMAND, ARGV(*)
19
          INTEGER MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
20
                     IERROR
21
          MPI_COMM_SPAWN tries to start maxprocs identical copies of the MPI program spec-
22
     ified by command, establishing communication with them and returning an intercommun-
23
     icator. The spawned processes are referred to as children. The children have their own
^{24}
     MPI_COMM_WORLD, which is separate from that of the parents. MPI_COMM_SPAWN is
25
     collective over comm, and also may not return until MPI INIT has been called in the chil-
26
     dren. Similarly, MPI_INIT in the children may not return until all parents have called
27
     MPI_COMM_SPAWN. In this sense, MPI_COMM_SPAWN in the parents and MPI_INIT in
28
     the children form a collective operation over the union of parent and child processes. The
29
     intercommunicator returned by MPI_COMM_SPAWN contains the parent processes in the
30
     local group and the child processes in the remote group. The ordering of processes in the
^{31}
     local and remote groups is the same as the ordering of the group of the comm in the parents
32
     and of MPI_COMM_WORLD of the children, respectively. This intercommunicator can be
33
     obtained in the children through the function MPI_COMM_GET_PARENT.
34
35
           Advice to users.
                             An implementation may automatically establish communication
36
          before MPI_INIT is called by the children. Thus, completion of MPI_COMM_SPAWN
37
          in the parent does not necessarily mean that MPI_INIT has been called in the children
38
           (although the returned intercommunicator can be used immediately). (End of advice
39
           to users.)
40
41
     The command argument The command argument is a string containing the name of a pro-
42
     gram to be spawned. The string is null-terminated in C. In Fortran, leading and trailing
43
     spaces are stripped. MPI does not specify how to find the executable or how the working
44
     directory is determined. These rules are implementation-dependent and should be appro-
45
     priate for the runtime environment.
46
47
           Advice to implementors. The implementation should use a natural rule for finding
```

Unofficial Draft for Comment Only

executables and determining working directories. For instance, a homogeneous system

with a global file system might look first in the working directory of the spawning process, or might search the directories in a PATH environment variable as do Unix shells. An implementation should document its rules for finding executables and determining working directories, and a high-quality implementation should give the user some control over these rules. (*End of advice to implementors.*)

If the program named in **command** does not call MPI_INIT, but instead forks a process that calls MPI_INIT, the results are undefined. Implementations may allow this case to work but are not required to.

Advice to users. MPI does not say what happens if the program you start is a shell script and that shell script starts a program that calls MPI_INIT. Though some implementations may allow you to do this, they may also have restrictions, such as requiring that arguments supplied to the shell script be supplied to the program, or requiring that certain parts of the environment not be changed. (*End of advice to users.*)

The argv argument argv is an array of strings containing arguments that are passed to the program. The first element of argv is the first argument passed to command, not, as is conventional in some contexts, the command itself. The argument list is terminated by NULL in C and an empty string in Fortran. In Fortran, leading and trailing spaces are always stripped, so that a string consisting of all spaces is considered an empty string. The constant MPI_ARGV_NULL may be used in C and Fortran to indicate an empty argument list. In C this constant is the same as NULL.

```
Example 11.17 Examples of argv in C and Fortran
To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" in C:
    char command[] = "ocean";
    char *argv[] = {"-gridfile", "ocean1.grd", NULL};
```

MPI_Comm_spawn(command, argv, ...);

or, if not everything is known at compile time:

```
char *command;
char **argv;
command = "ocean";
argv=(char **)malloc(3 * sizeof(char *));
argv[0] = "-gridfile";
argv[1] = "ocean1.grd";
argv[2] = NULL;
MPI_Comm_spawn(command, argv, ...);
```

In Fortran:

```
      CHARACTER*25 command, argv(3)
      43

      command = 'ocean'
      44

      argv(1) = '-gridfile'
      45

      argv(2) = 'ocean1.grd'
      46

      argv(3) = ','
      47

      call MPI_COMM_SPAWN(command, argv, ...)
      48
```

 $\mathbf{2}$

480 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1 Arguments are supplied to the program if this is allowed by the operating system. In $\mathbf{2}$ C, the MPI_COMM_SPAWN argument argv differs from the argv argument of main in two 3 respects. First, it is shifted by one element. Specifically, argv[0] of main is provided by the 4 implementation and conventionally contains the name of the program (given by **command**). $\mathbf{5}$ argv[1] of main corresponds to argv[0] in MPI_COMM_SPAWN, argv[2] of main to argv[1] 6 of MPI_COMM_SPAWN, etc. Passing an argv of MPI_ARGV_NULL to MPI_COMM_SPAWN $\overline{7}$ results in main receiving argc of 1 and an argv whose element 0 is (conventionally) the 8 name of the program. Second, argv of MPI_COMM_SPAWN must be null-terminated, so 9 that its length can be determined.

¹⁰ If a Fortran implementation supplies routines that allow a program to obtain its ar-¹¹ guments, the arguments may be available through that mechanism. In C, if the operating ¹² system does not support arguments appearing in argv of main(), the MPI implementation ¹³ may add the arguments to the argv that is passed to MPI_INIT.

14

¹⁵ The maxprocs argument MPI tries to spawn maxprocs processes. If it is unable to spawn ¹⁶ maxprocs processes, it raises an error of class MPI_ERR_SPAWN.

¹⁷ An implementation may allow the info argument to change the default behavior, such ¹⁸ that if the implementation is unable to spawn all maxprocs processes, it may spawn a ¹⁹ smaller number of processes instead of raising an error. In principle, the info argument ²⁰ may specify an arbitrary set $\{m_i : 0 \le m_i \le \text{maxprocs}\}$ of allowed values for the number ²¹ of processes spawned. The set $\{m_i\}$ does not necessarily include the value maxprocs. If ²² an implementation is able to spawn one of these allowed numbers of processes,

²³ MPI_COMM_SPAWN returns successfully and the number of spawned processes, *m*, is given ²⁴ by the size of the remote group of intercomm. If *m* is less than maxproc, reasons why the ²⁵ other processes were not spawned are given in array_of_errcodes as described below. If it is ²⁶ not possible to spawn one of the allowed numbers of processes, MPI_COMM_SPAWN raises ²⁷ an error of class MPI_ERR_SPAWN.

A spawn call with the default behavior is called *hard*. A spawn call for which fewer than
 maxprocs processes may be returned is called "*soft*". See Section 11.8.4 for more information
 on the "soft" key for info.

Advice to users. By default, requests are hard and MPI errors are fatal. This means that by default there will be a fatal error if MPI cannot spawn all the requested processes. If you want the behavior "spawn as many processes as possible, up to N," you should do a soft spawn, where the set of allowed values $\{m_i\}$ is $\{0, \ldots, N\}$. However, this is not completely portable, as implementations are not required to support soft spawning. (End of advice to users.)

- The info argument The info argument to all of the routines in this chapter is an opaque handle of type MPI_Info in C and Fortran with the mpi_f08 module and INTEGER in Fortran with the mpi module or the include file mpif.h. It is a container for a number of user-specified (key,value) pairs. key and value are strings (null-terminated char* in C, character*(*) in Fortran). Routines to create and manipulate the info argument are described in Chapter 10.
- For the SPAWN calls, info provides additional (and possibly implementation-dependent)
 instructions to MPI and the runtime system on how to start processes. An application may
 pass MPI_INFO_NULL in C or Fortran. Portable programs not requiring detailed control over
 process locations should use MPI_INFO_NULL.
- 48

32

33

34

35

36

37

MPI does not specify the content of the info argument, except to reserve a number of special key values (see Section 11.8.4). The info argument is quite flexible and could even be used, for example, to specify the executable and its command-line arguments. In this case the command argument to MPI_COMM_SPAWN could be empty. The ability to do this follows from the fact that MPI does not specify how an executable is found, and the info argument can tell the runtime system where to "find" the executable "" (empty string). Of course a program that does this will not be portable across MPI implementations.

The root argument All arguments before the root argument are examined only on the process whose rank in comm is equal to root. The value of these arguments on other processes is ignored.

The array_of_errcodes argument The array_of_errcodes is an array of length maxprocs in which MPI reports the status of each process that MPI was requested to start. If all maxprocs processes were spawned, array_of_errcodes is filled in with the value MPI_SUCCESS. If only m ($0 \le m < \text{maxprocs}$) processes are spawned, m of the entries will contain MPI_SUCCESS and the rest will contain an implementation-specific error code indicating the reason MPI could not start the process. MPI does not specify which entries correspond to failed processes. An implementation may, for instance, fill in error codes in one-to-one correspondence with a detailed specification in the info argument. These error codes all belong to the error class MPI_ERR_SPAWN if there was no error in the argument list. In C or Fortran, an application may pass MPI_ERRCODES_IGNORE if it is not interested in the error codes.

Advice to implementors. MPI_ERRCODES_IGNORE in Fortran is a special type of constant, like MPI_BOTTOM. See the discussion in Section 2.5.4. (*End of advice to implementors.*)

MPI_COMM_GET_PARENT(parent)	29
	30
OUT parent the parent communicator (handle)	31
	32
C binding	33
<pre>int MPI_Comm_get_parent(MPI_Comm *parent)</pre>	34
Fortran 2008 binding	35
5	36
MPI_Comm_get_parent(parent, ierror)	37
TYPE(MPI_Comm), INTENT(OUT) :: parent	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
	39
Fortran binding	40
MPI_COMM_GET_PARENT(PARENT, IERROR)	41
INTEGER PARENT, IERROR	42
If a pressed was started with MDL COMM SDAWN or MDL COMM SDAWN MULTIPLE	43
If a process was started with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE,	44
MPI_COMM_GET_PARENT returns the "parent" intercommunicator of the current process.	45
This parent intercommunicator is created implicitly inside of MPI_INIT and is the same in-	

tercommunicator returned by SPAWN in the parents.

If the process was not spawned, MPI_COMM_GET_PARENT returns MPI_COMM_NULL.

 $\mathbf{2}$

 24

_			
1 2		-	s freed or disconnected, MPI_COMM_GET_PARENT
2	returns M	IPI_COMM_NULL.	
4	Ada	nice to users MPL COMM	GET_PARENT returns a handle to a single intercom-
5			A_{GET} PARENT a second time returns a handle to
6		0	eeing the handle with MPI_COMM_DISCONNECT or
7			ther references to the intercommunicator to become
8			ling MPI_COMM_FREE on the parent communicator
9		ot useful. (End of advice to	
10	15 11	of userul. (End of uddice to	
11	Rat	<i>ionale.</i> The desire of the	e Forum was to create a constant
12	MPI	_COMM_PARENT similar to	MPI_COMM_WORLD. Unfortunately such a constant
12			s an argument to MPI_COMM_DISCONNECT, which
14		xplicitly allowed. (End of rat	, , , , , , , , , , , , , , , , , , ,
15			,
16	11.8.3	Starting Multiple Executable	s and Establishing Communication
17			
18			ent for most cases, it does not allow the spawning
19	-	,	binary with multiple sets of arguments. The follow-
20	-		or the same binary with multiple sets of arguments,
21	establishi	ng communication with then	and placing them in the same MPI_COMM_WORLD.
22			
23		AM SPAWN MULTIPLE(cou	nt, array_of_commands, array_of_argv,
24			ray_of_info, root, comm, intercomm,
25		array_of_errcodes)	
26		- , , , , , , , , , , , , , , , , , , ,	
27	IN	count	number of commands (positive integer, significant
28			only at root)
29	IN	array_of_commands	programs to be executed (array of strings, significant
30			only at root)
31	IN	array_of_argv	arguments for commands (array of array of strings,
32		,	significant only at root)
33	IN	array of mayness	,
34	IIN	array_of_maxprocs	maximum number of processes to start for each
35			command (array of integers, significant only at root)
36	IN	array_of_info	info objects telling the runtime system where and
37			how to start processes (array of handles, significant
38			only at root)
39	IN	root	rank of process in which previous arguments are
40			examined (integer)
41	IN	comm	intracommunicator containing group of growning
42	IIN	comm	intracommunicator containing group of spawning processes (handle)
43			
44	OUT	intercomm	intercommunicator between original group and the
45			newly spawned group (handle)
46	OUT	array_of_errcodes	one error code per process (array of integers)
47		-	v v v ,
48	C bindir	ופ	

⁴⁸ C binding

```
1
int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
                                                                                    \mathbf{2}
              char **array_of_argv[], const int array_of_maxprocs[],
                                                                                    3
              const MPI_Info array_of_info[], int root, MPI_Comm comm,
              MPI_Comm *intercomm, int array_of_errcodes[])
                                                                                    4
Fortran 2008 binding
                                                                                    6
MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
              array_of_maxprocs, array_of_info, root, comm, intercomm,
              array_of_errcodes, ierror)
                                                                                    9
    INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
                                                                                   10
    CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
                                                                                   11
               array_of_argv(count, *)
                                                                                   12
    TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
                                                                                   13
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   14
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
                                                                                   15
    INTEGER :: array_of_errcodes(*)
                                                                                   16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   17
                                                                                   18
Fortran binding
                                                                                   19
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
                                                                                   20
              ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
                                                                                   21
              ARRAY_OF_ERRCODES, IERROR)
    INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*), ROOT, COMM,
                                                                                   22
                                                                                   23
               INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
                                                                                   24
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
                                                                                   25
```

MPI_COMM_SPAWN_MULTIPLE is identical to MPI_COMM_SPAWN except that there are multiple executable specifications. The first argument, count, gives the number of specifications. Each of the next four arguments are simply arrays of the corresponding arguments in MPI_COMM_SPAWN. For the Fortran version of array_of_argv, the element array_of_argv(i,j) is the j-th argument to command number i.

Rationale. This may seem backwards to Fortran programmers who are familiar with Fortran's column-major ordering. However, it is necessary to do it this way to allow MPI_COMM_SPAWN to sort out arguments. Note that the leading dimension of array_of_argv *must* be the same as count. Also note that Fortran rules for sequence association allow a different value in the first dimension; in this case, the sequence of array elements is interpreted by MPI_COMM_SPAWN_MULTIPLE as if the sequence is stored in an array defined with the first dimension set to count. This Fortran feature allows an implementor to define MPI_ARGVS_NULL (see below) with fixed dimensions, e.g., (1,1), or only with one dimension, e.g., (1). (*End of rationale.*)

Advice to users. The argument count is interpreted by MPI only at the root, as is array_of_argv. Since the leading dimension of array_of_argv is count, a non-positive value of count at a non-root node could theoretically cause a runtime bounds check error, even though array_of_argv should be ignored by the subroutine. If this happens, you should explicitly supply a reasonable value of count on the non-root nodes. (*End of advice to users.*)

In any language, an application may use the constant MPI_ARGVS_NULL (which is likely to be (char ***)0 in C) to specify that no arguments should be passed to any commands.

Unofficial Draft for Comment Only

26

27

28

29

30

31

32

33

34

35

36

37

38

39 40

41

42

43

44

45

 $46 \\ 47$

1 The effect of setting individual elements of array_of_argv to MPI_ARGV_NULL is not defined. $\mathbf{2}$ To specify arguments for some commands but not others, the commands without arguments 3 should have a corresponding argv whose first element is null ((char *)0 in C and empty 4 string in Fortran). In Fortran at non-root processes, the count argument must be set to $\mathbf{5}$ a value that is consistent with the provided array_of_argv although the content of these 6 arguments has no meaning for this operation.

7All of the spawned processes have the same MPI_COMM_WORLD. Their ranks in 8 MPI_COMM_WORLD correspond directly to the order in which the commands are specified 9 in MPI_COMM_SPAWN_MULTIPLE. Assume that m_1 processes are generated by the first 10 command, m_2 by the second, etc. The processes corresponding to the first command have 11ranks $0, 1, \ldots, m_1 - 1$. The processes in the second command have ranks $m_1, m_1 + 1, \ldots, m_1 + 1$ 12 $m_2 - 1$. The processes in the third have ranks $m_1 + m_2, m_1 + m_2 + 1, \ldots, m_1 + m_2 + m_3 - 1$, 13etc.

1415

16

18

21

25

30

Calling MPI_COMM_SPAWN multiple times would create many Advice to users. sets of children with different MPI_COMM_WORLDs whereas

MPI_COMM_SPAWN_MULTIPLE creates children with a single MPI_COMM_WORLD, 17 so the two methods are not completely equivalent. There are also two performancerelated reasons why, if you need to spawn multiple executables, you may want to 19 use MPI_COMM_SPAWN_MULTIPLE instead of calling MPI_COMM_SPAWN several 20times. First, spawning several things at once may be faster than spawning them sequentially. Second, in some implementations, communication between processes 22 spawned at the same time may be faster than communication between processes 23spawned separately. (End of advice to users.) 24

The array_of_errcodes argument is a 1-dimensional array of size $\sum_{i=1}^{count} n_i$, where n_i is 26the *i*-th element of array_of_maxprocs. Command number *i* corresponds to the n_i contiguous 27slots in this array from element $\sum_{j=1}^{i-1} n_j$ to $\left[\sum_{j=1}^{i} n_j\right] - 1$. Error codes are treated as for 28MPI_COMM_SPAWN. 29

Example 11.18 Examples of array_of_argv in C and Fortran 31 To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" and the program 32 "atmos" with argument "atmos.grd" in C: 33

```
34
             char *array_of_commands[2] = {"ocean", "atmos"};
35
             char **array_of_argv[2];
36
             char *argv0[] = {"-gridfile", "ocean1.grd", (char *)0};
37
             char *argv1[] = {"atmos.grd", (char *)0};
38
             array_of_argv[0] = argv0;
39
             array_of_argv[1] = argv1;
40
             MPI_Comm_spawn_multiple(2, array_of_commands, array_of_argv, ...);
41
42
     Here is how you do it in Fortran:
43
44
45
46
47
48
```

```
CHARACTER*25 commands(2), array_of_argv(2, 3)

commands(1) = 'ocean'

array_of_argv(1, 1) = '-gridfile'

array_of_argv(1, 2) = 'ocean1.grd'

array_of_argv(1, 3) = ' '

commands(2) = 'atmos'

array_of_argv(2, 1) = 'atmos.grd'

array_of_argv(2, 2) = ' '

call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)

11.8.4 Reserved Keys

The following keys are reserved. An implementation is not required to interpret these keys,

but if it does interpret the key, it must provide the functionality described.
```

- "host" Value is a hostname. The format of the hostname is determined by the implementation.
- "arch" Value is an architecture name. Valid architecture names and what they mean are determined by the implementation.
- "wdir" Value is the name of a directory on a machine on which the spawned process(es) execute(s). This directory is made the working directory of the executing process(es). The format of the directory name is determined by the implementation.
- "**path**" Value is a directory or set of directories where the implementation should look for the executable. The format of "**path**" is determined by the implementation.
- "file" Value is the name of a file in which additional information is specified. The format of the filename and internal format of the file are determined by the implementation.
- "mpi_initial_errhandler" Value is the name of an errhandler that will be set as the initial error handler. The "mpi_initial_errhandler" key can take the case insensitive values "mpi_errors_are_fatal", "mpi_errors_abort", and "mpi_errors_return" representing the predefined MPI error handlers (MPI_ERRORS_ARE_FATAL—the default, MPI_ERRORS_ABORT, and MPI_ERRORS_RETURN, respectively). Other, non-standard values may be supported by the implementation, which should document the resultant behavior.
- "soft" Value specifies a set of numbers which are allowed values for the number of processes that MPI_COMM_SPAWN (et al.) may create. The format of the value is a commaseparated list of Fortran-90 triplets each of which specifies a set of integers and which together specify the set formed by the union of these sets. Negative values in this set and values greater than maxprocs are ignored. MPI will spawn the largest number of processes it can, consistent with some number in the set. The order in which triplets are given is not significant.

By Fortran-90 triplets, we mean:

1. a means a

1

2

5 6

10

11 12 13

14

15

16 17

18

19 20

21

22

23

24

25 26

27

28

29

30 31

32

33

34

35

36

37

38

46

486 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

```
1
            2. a:b means a, a + 1, a + 2, ..., b
2
            3. a:b:c means a, a + c, a + 2c, \ldots, a + ck, where for c > 0, k is the largest integer
3
               for which a + ck < b and for c < 0, k is the largest integer for which a + ck > b.
4
               If b > a then c must be positive. If b < a then c must be negative.
5
6
          Examples:
7
            1. a:b gives a range between a and b
8
            2. O:N gives full "soft" functionality
9
10
            3. 1,2,4,8,16,32,64,128,256,512,1024,2048,4096 allows a power-of-two num-
11
               ber of processes.
12
            4. 2:10000:2 allows an even number of processes.
13
            5. 2:10:2,7 allows 2, 4, 6, 7, 8, or 10 processes.
14
15
     11.8.5 Spawn Example
16
17
18
     Example 11.19 Manager-worker Example Using MPI_COMM_SPAWN
19
20
     /* manager */
21
     #include "mpi.h"
22
     int main(int argc, char *argv[])
23
     {
24
        int world_size, universe_size, *universe_sizep, flag;
25
        MPI_Comm everyone;
                                         /* intercommunicator */
26
        char worker_program[100];
27
28
        MPI_Init(&argc, &argv);
29
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);
30
        if (world_size != 1)
31
                                   error("Top heavy with management");
32
33
        MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
34
                            &universe_sizep, &flag);
35
        if (!flag) {
36
              printf("This MPI does not support UNIVERSE_SIZE. How many\n\
37
     processes total?");
38
              scanf("%d", &universe_size);
39
        } else universe_size = *universe_sizep;
40
        if (universe_size == 1) error("No room to start workers");
41
42
        /*
43
         * Now spawn the workers. Note that there is a run-time determination
44
         * of what type of worker to spawn, and presumably this calculation must
45
         * be done at run time and cannot be calculated before starting
46
         * the program. If everything is known when the application is
47
         * first started, it is generally better to start them all at once
48
         * in a single MPI_COMM_WORLD.
```

```
1
    */
                                                                                      2
   choose_worker_program(worker_program);
   MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
              MPI_INFO_NULL, 0, MPI_COMM_SELF, & everyone,
                                                                                      5
              MPI_ERRCODES_IGNORE);
                                                                                      6
   /*
    * Parallel code here. The communicator "everyone" can be used
    * to communicate with the spawned processes, which have ranks 0,...
                                                                                      9
                                                                                      10
    * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
                                                                                      11
    * "everyone".
    */
                                                                                      12
                                                                                      13
                                                                                      14
   MPI_Finalize();
                                                                                      15
   return 0;
                                                                                      16
}
                                                                                      17
/* worker */
                                                                                      18
                                                                                      19
#include "mpi.h"
                                                                                      20
int main(int argc, char *argv[])
                                                                                      21
{
                                                                                      22
   int size;
                                                                                      23
   MPI_Comm parent;
                                                                                      24
   MPI_Init(&argc, &argv);
                                                                                      25
   MPI_Comm_get_parent(&parent);
                                                                                      26
   if (parent == MPI_COMM_NULL) error("No parent!");
                                                                                      27
   MPI_Comm_remote_size(parent, &size);
                                                                                      28
   if (size != 1) error("Something's wrong with the parent");
                                                                                      29
                                                                                      30
   /*
                                                                                      31
    * Parallel code here.
                                                                                      32
    \ast The manager is represented as the process with rank 0 in (the remote
                                                                                      33
    * group of) the parent communicator. If the workers need to communicate
                                                                                      34
    * among themselves, they can use MPI_COMM_WORLD.
                                                                                      35
    */
                                                                                      36
                                                                                      37
   MPI_Finalize();
                                                                                      38
   return 0:
                                                                                      39
}
                                                                                      40
                                                                                      41
                                                                                      42
                                                                                      43
                                                                                      44
11.9
       Establishing Communication
                                                                                      45
                                                                                      46
This section provides functions that establish communication between two sets of MPI
                                                                                      47
processes that do not share a communicator.
```

Some situations in which these functions are useful are:

Unofficial Draft for Comment Only

- 1. Two parts of an application that are started independently need to communicate.
 - 2. A visualization tool wants to attach to a running process.
 - 3. A server wants to accept connections from multiple clients. Both clients and server may be parallel programs.

In each of these situations, MPI must establish communication channels where none existed 8 before, and there is no parent/child relationship. The routines described in this section 9 establish communication between the two sets of processes by creating an MPI intercom-10 municator, where the two groups of the intercommunicator are the original sets of processes. 11 Establishing contact between two groups of processes that do not share an existing communicator is a collective but asymmetric process. One group of processes indicates its 12willingness to accept connections from other groups of processes. We will call this group 13 the (parallel) server, even if this is not a client/server type of application. The other group 14

connects to the server; we will call it the *client*. 15

Advice to users. While the names *client* and *server* are used throughout this section, MPI does not guarantee the traditional robustness of client/server systems. The functionality described in this section is intended to allow two cooperating parts of the same application to communicate with one another. For instance, a client that gets a segmentation fault and dies, or one that does not participate in a collective operation may cause a server to crash or hang. (End of advice to users.)

22 23 24

25

26

27

28

29

30

 31

32

33

34

35 36

37

38

39 40

41

4243

44

45

16

17

18

19

20

21

1

2

3 4

5

6 $\overline{7}$

> Names, Addresses, Ports, and All That 11.9.1

Almost all of the complexity in MPI client/server routines addresses the question "how does the client find out how to contact the server?" The difficulty, of course, is that there is no existing communication channel between them, yet they must somehow agree on a rendezvous point where they will establish communication.

Agreeing on a rendezvous point always involves a third party. The third party may itself provide the rendezvous point or may communicate rendezvous information from server to client. Complicating matters might be the fact that a client does not really care what server it contacts, only that it be able to get in touch with one that can handle its request.

Ideally, MPI can accommodate a wide variety of run-time systems while retaining the ability to write simple, portable code. The following should be compatible with MPI:

- The server resides at a well-known internet address host:port.
- The server prints out an address to the terminal; the user gives this address to the client program.
- The server places the address information on a nameserver, where it can be retrieved with an agreed-upon name.
- The server to which the client connects is actually a broker, acting as a middleman between the client and the real server.

46 MPI does not require a nameserver, so not all implementations will be able to support 47all of the above scenarios. However, MPI provides an optional nameserver interface, and is 48compatible with external name servers.

A port_name is a *system-supplied* string that encodes a low-level network address at which a server can be contacted. Typically this is an IP address and a port number, but an implementation is free to use any protocol. The server establishes a port_name with the MPI_OPEN_PORT routine. It accepts a connection to a given port with MPI_COMM_ACCEPT. A client uses port_name to connect to the server.

By itself, the port_name mechanism is completely portable, but it may be clumsy to use because of the necessity to communicate port_name to the client. It would be more convenient if a server could specify that it be known by an *application-supplied* service_name so that the client could connect to that service_name without knowing the port_name.

An MPI implementation may allow the server to publish a (port_name, service_name) pair with MPI_PUBLISH_NAME and the client to retrieve the port name from the service name with MPI_LOOKUP_NAME. This allows three levels of portability, with increasing levels of functionality.

- 1. Applications that do not rely on the ability to publish names are the most portable. Typically the port_name must be transferred "by hand" from server to client.
- 2. Applications that use the MPI_PUBLISH_NAME mechanism are completely portable among implementations that provide this service. To be portable among all implementations, these applications should have a fall-back mechanism that can be used when names are not published.
- 3. Applications may ignore MPI's name publishing functionality and use their own mechanism (possibly system-supplied) to publish names. This allows arbitrary flexibility but is not portable.

11.9.2 Server Routines

A server makes itself available with two routines. First it must call MPI_OPEN_PORT to establish a port at which it may be contacted. Secondly it must call MPI_COMM_ACCEPT to accept connections from clients.

MPI_OPEN_PORT(info, port_name)

IN	info	implementation-specific information on how to establish an address (handle)
OUT	port_name	newly established port (string)
C binding		
int MPI_0	pen_port(MPI_Info info, c	char *port_name)

Fortran 2008 binding

```
MPI_Open_port(info, port_name, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)

 $\mathbf{2}$

 $\overline{7}$

 $45 \\ 46$

1 2	INTEGER INFO, IERROR CHARACTER*(*) PORT_NAME
3 4 5 6 7 8 9	This function establishes a network address, encoded in the port_name string, at which the server will be able to accept connections from clients. port_name is supplied by the system, possibly using information in the info argument. MPI copies a system-supplied port name into port_name. port_name identifies the newly opened port and can be used by a client to contact the server. The maximum size string that may be supplied by the system is MPI_MAX_PORT_NAME.
10 11 12	Advice to users. The system copies the port name into port_name. The application must pass a buffer of sufficient size to hold this value. (<i>End of advice to users.</i>)
12 13 14 15 16 17	port_name is essentially a network address. It is unique within the communication universe to which it belongs (determined by the implementation), and may be used by any client within that communication universe. For instance, if it is an internet (host:port) address, it will be unique on the internet. If it is a low level switch address on an IBM SP, it will be unique to that SP.
18 19 20 21 22 23	Advice to implementors. These examples are not meant to constrain implementa- tions. A port_name could, for instance, contain a user name or the name of a batch job, as long as it is unique within some well-defined communication domain. The larger the communication domain, the more useful MPI's client/server functionality will be. (End of advice to implementors.)
24 25 26 27 28	The precise form of the address is implementation-defined. For instance, an internet address may be a host name or IP address, or anything that the implementation can decode into an IP address. A port name may be reused after it is freed with MPI_CLOSE_PORT and released by the system.
29 30 31 32	Advice to implementors. Since the user may type in port_name by hand, it is useful to choose a form that is easily readable and does not have embedded spaces. (End of advice to implementors.)
33 34 35 36	info may be used to tell the implementation how to establish the address. It may, and usually will, be MPI_INFO_NULL in order to get the implementation defaults.
37 38	MPI_CLOSE_PORT(port_name) IN port_name a port (string)
39 40 41	C binding int MPI_Close_port(const char *port_name)
42 43 44 45 46	<pre>Fortran 2008 binding MPI_Close_port(port_name, ierror) CHARACTER(LEN=*), INTENT(IN) :: port_name INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
47 48	Fortran binding MPI_CLOSE_PORT(PORT_NAME, IERROR)

	HARACTER*(*) PORT_NAME NTEGER IERROR		1 2
		etwork address represented by port_name.	3 4
			5
MPI_0	COMM_ACCEPT(port_nam	e, info, root, comm, newcomm)	6
IN	port_name	port name (string, significant only at root)	7
IN	info	implementation-dependent information (handle,	8 9
IIN	inio	significant only at root)	10
IN	root	rank in comm of root node (integer)	11 12
IN	comm	intracommunicator over which call is collective (handle)	12 13 14
τυο	newcomm	intercommunicator with client as remote group (handle)	15 16 17
C bin	ding		18
	PI_Comm_accept(const c	har *port_name, MPI_Info info, int root, MPI_Comm *newcomm)	19 20
D (21
	an 2008 binding	info, root, comm, newcomm, ierror)	22
	HARACTER(LEN=*), INTEN		23 24
	YPE(MPI_Info), INTENT(-	24
INTEGER, INTENT(IN) :: root			26
TYPE(MPI_Comm), INTENT(IN) :: comm			27
	YPE(MPI_Comm), INTENT(28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			29
Fortr	an binding		30
MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)			31 32
	HARACTER*(*) PORT_NAME		33
1	NTEGER INFO, ROOT, COM	M, NEWCUMM, IERRUR	34
		blishes communication with a client. It is collective over the	35
-	communicator. It returns	an intercommunicator that allows communication with the	36
client.	1	and a thirth of the second of a second to MDL ODEN DODT	37
		een established through a call to MPI_OPEN_PORT. directives that may influence the behavior of the ACCEPT	38
call.	to can be used to provide	directives that may initiatice the behavior of the ACCLI	39 40
			40
11.9.3	Client Routines		42
Thore	is only one routing on the	aliant sida	43
THELE	is only one routine on the		44
			45
			46 47
			48

492 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1 MPI_COMM_CONNECT(port_name, info, root, comm, newcomm) 2 IN network address (string, significant only at root) port_name 3 IN info implementation-dependent information (handle, 4 significant only at root) 56 IN root rank in comm of root node (integer) 7 IN intracommunicator over which call is collective comm 8 (handle) 9 OUT newcomm intercommunicator with server as remote group 10 (handle) 11 1213 C binding 14int MPI_Comm_connect(const char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm) 1516Fortran 2008 binding 17MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror) 18 CHARACTER(LEN=*), INTENT(IN) :: port_name 19TYPE(MPI_Info), INTENT(IN) :: info 20INTEGER, INTENT(IN) :: root 21TYPE(MPI_Comm), INTENT(IN) :: comm 22 TYPE(MPI_Comm), INTENT(OUT) :: newcomm 23INTEGER, OPTIONAL, INTENT(OUT) :: ierror 24 25Fortran binding 26MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR) 27CHARACTER*(*) PORT_NAME INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR 2829 This routine establishes communication with a server specified by port_name. It is 30 collective over the calling communicator and returns an intercommunicator in which the 31 remote group participated in an MPI_COMM_ACCEPT. 32 If the named port does not exist (or has been closed), MPI_COMM_CONNECT raises 33 an error of class MPI_ERR_PORT. 34 If the port exists, but does not have a pending MPI_COMM_ACCEPT, the connection 35 attempt will eventually time out after an implementation-defined time, or succeed when 36 the server calls MPI_COMM_ACCEPT. In the case of a time out, MPI_COMM_CONNECT 37 raises an error of class MPI_ERR_PORT. 38 39 Advice to implementors. The time out period may be arbitrarily short or long. 40 However, a high-quality implementation will try to queue connection attempts so 41 that a server can handle simultaneous requests from several clients. A high-quality 42implementation may also provide a mechanism, through the info arguments to 43 MPI_OPEN_PORT, MPI_COMM_ACCEPT, and/or MPI_COMM_CONNECT, for the 44user to control timeout and queuing behavior. (End of advice to implementors.) 4546MPI provides no guarantee of fairness in servicing connection attempts. That is, connec-47tion attempts are not necessarily satisfied in the order they were initiated and competition 48

from other connection attempts may prevent a particular connection attempt from being satisfied.

port_name is the address of the server. It must be the same as the name returned by MPI_OPEN_PORT on the server. Some freedom is allowed here. If there are equivalent forms of port_name, an implementation may accept them as well. For instance, if port_name is (hostname:port), an implementation may accept (ip_address:port) as well.

11.9.4 Name Publishing

The routines in this section provide a mechanism for publishing names. A (service_name, port_name) pair is published by the server, and may be retrieved by a client using the service_name only. An MPI implementation defines the *scope* of the service_name, that is, the domain over which the service_name can be retrieved. If the domain is the empty set, that is, if no client can retrieve the information, then we say that name publishing is not supported. Implementations should document how the scope is determined. High-quality implementations will give some control to users through the info arguments to name publishing functions. Examples are given in the descriptions of individual functions.

MDI DUDUCH NAME(comico nomo info nort nomo)				
MPI_PUBLISH_NAME(service_name, info, port_name)			20	
IN	service_name	a service name to associate with the port (string)	21	
IN	info	implementation-specific information (handle)	22	
IN	port_name	a port name (string)	23	
			24	
C binding	y		25	
		service name MPT Info info	26	
IIIC MII_I	<pre>int MPI_Publish_name(const char *service_name, MPI_Info info,</pre>			
	const char *port_name)			
Fortran 2008 binding				
MPI_Publi	MPI_Publish_name(service_name, info, port_name, ierror)			
CHARA	CTER(LEN=*), INTENT(IN)	:: service_name, port_name	31	
TYPE(TYPE(MPI_Info), INTENT(IN) :: info			
INTEG	ER, OPTIONAL, INTENT(OUT) :: ierror	33	
D ()			34	
	Fortran binding			
	SH_NAME(SERVICE_NAME, IN		36	
	CTER*(*) SERVICE_NAME, F	PORT_NAME	37	
INTEG	ER INFO, IERROR		38	
This r	outine publishes the pair (po	ort_name, service_name) so that an application may	39	

This routine publishes the pair (port_name, service_name) so that an application may retrieve a system-supplied port_name using a well-known service_name.

The implementation must define the *scope* of a published service name, that is, the domain over which the service name is unique, and conversely, the domain over which the (port name, service name) pair may be retrieved. For instance, a service name may be unique to a job (where job is defined by a distributed operating system or batch scheduler), unique to a machine, or unique to a Kerberos realm. The scope may depend on the info argument to MPI_PUBLISH_NAME.

MPI permits publishing more than one service_name for a single port_name. On the ⁴⁷ other hand, if service_name has already been published within the scope determined by info, ⁴⁸

Unofficial Draft for Comment Only

 $45 \\ 46$

the behavior of MPI_PUBLISH_NAME is undefined. An MPI implementation may, through
 a mechanism in the info argument to MPI_PUBLISH_NAME, provide a way to allow multiple
 servers with the same service in the same scope. In this case, an implementation-defined
 policy will determine which of several port names is returned by MPI_LOOKUP_NAME.

Note that while service_name has a limited scope, determined by the implementation,
 port_name always has global scope within the communication universe used by the imple mentation (i.e., it is globally unique).

port_name should be the name of a port established by MPI_OPEN_PORT and not yet
 released by MPI_CLOSE_PORT. If it is not, the result is undefined.

Advice to implementors. In some cases, an MPI implementation may use a name service that a user can also access directly. In this case, a name published by MPI could easily conflict with a name published by a user. In order to avoid such conflicts, MPI implementations should mangle service names so that they are unlikely to conflict with user code that makes use of the same service. Such name mangling will of course be completely transparent to the user.

¹⁷ The following situation is problematic but unavoidable, if we want to allow implemen-¹⁸ tations to use nameservers. Suppose there are multiple instances of "ocean" running ¹⁹ on a machine. If the scope of a service name is confined to a job, then multiple ²⁰ oceans can coexist. If an implementation provides site-wide scope, however, multiple ²¹ instances are not possible as all calls to MPI_PUBLISH_NAME after the first may fail. ²² There is no universal solution to this.

To handle these situations, a high-quality implementation should make it possible to limit the domain over which names are published. (*End of advice to implementors.*)

```
25
26
27
```

32

33 34

35

36

37

42

10

11

12

13

14

15

16

MPI_UNPUBLISH_NAME(service_name, info, port_name)
 IN service_name a service name (string)
 IN info implementation-specific information (handle)

IN port_name

a port name (string)

C binding

```
<sup>38</sup> Fortran 2008 binding
```

```
MPI_Unpublish_name(service_name, info, port_name, ierror)
CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
TYPE(MPI_Info), INTENT(IN) :: info
```

- INTEGER, OPTIONAL, INTENT(OUT) :: ierror
- 43 44 Fortran binding

```
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)

CHARACTER*(*) SERVICE_NAME, PORT_NAME

INTEGER INFO, IERROR
```

This routine unpublishes a service name that has been previously published. Attempting to unpublish a name that has not been published or has already been unpublished is erroneous and is indicated by the error class MPI_ERR_SERVICE.

All published names must be unpublished before the corresponding port is closed and before the publishing process exits. The behavior of MPI_UNPUBLISH_NAME is implementation dependent when a process tries to unpublish a name that it did not publish.

If the info argument was used with MPI_PUBLISH_NAME to tell the implementation how to publish names, the implementation may require that info passed to MPI_UNPUBLISH_NAME contain information to tell the implementation how to unpublish a name.

MPI_LOO	<pre>KUP_NAME(service_name, info</pre>	o, port_name)	13
IN	service_name	a service name (string)	14
IN	info	implementation-specific information (handle)	15
OUT	port_name	a port name (string)	16 17
001	port_name	a poro indire (coring)	18
C binding	g		19
		ervice_name, MPI_Info info,	20
	char *port_name)		21
Fortran 2	008 binding		22
	p_name(service_name, info	, port_name, ierror)	23
	CTER(LEN=*), INTENT(IN) :	-	24 25
TYPE(MPI_Info), INTENT(IN) ::	info	23 26
		ME), INTENT(OUT) :: port_name	27
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	28
Fortran b	binding		29
MPI_LOOKU	MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)		
	CTER*(*) SERVICE_NAME, PC	IRT_NAME	31
INTEG	ER INFO, IERROR		32
This f	unction retrieves a port_name	published by MPI_PUBLISH_NAME with	$33 \\ 34$
service_nar	ne. If service_name has not b	een published, it raises an error in the error class	35
		upply a port_name buffer large enough to hold the	36
· ·	-	n above under MPI_OPEN_PORT).	37
		le entries with the same service_name within the	38
same scope	e, a particular port_name is ch	osen in a way determined by the implementation.	39

same scope, a particular port_name is chosen in a way determined by the implementation. If the info argument was used with MPI_PUBLISH_NAME to tell the implementation how to publish names, a similar info argument may be required for MPI_LOOKUP_NAME.

11.9.5 Reserved Key Values

The following key values are reserved. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described.

"ip_port" Value contains IP port number at which to establish a port. (Reserved for MPI_OPEN_PORT only).

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8 9

10

11 12

40

41 42

43 44

45

 $46 \\ 47$

```
1
     "ip_address" Value contains IP address at which to establish a port. If the address is not a
\mathbf{2}
          valid IP address of the host on which the MPI_OPEN_PORT call is made, the results
3
          are undefined. (Reserved for MPI_OPEN_PORT only).
4
5
     11.9.6 Client/Server Examples
6
7
     Example 11.20 Simplest Example — Completely Portable.
8
     The following example shows the simplest way to use the client/server interface. It does
9
     not use service names at all.
10
     On the server side:
11
12
13
         char myport[MPI_MAX_PORT_NAME];
14
         MPI_Comm intercomm;
15
         /* ... */
16
         MPI_Open_port(MPI_INFO_NULL, myport);
17
         printf("port name is: %s\n", myport);
18
19
         MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
20
         /* do something with intercomm */
21
     The server prints out the port name to the terminal and the user must type it in when
22
     starting up the client (assuming the MPI implementation supports stdin such that this
23
     works). On the client side:
24
25
         MPI_Comm intercomm;
26
          char name[MPI_MAX_PORT_NAME];
27
         printf("enter port name: ");
28
         gets(name);
29
         MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
30
31
     Example 11.21 Ocean/Atmosphere — Relies on Name Publishing
32
     In this example, the "ocean" application is the "server" side of a coupled ocean-atmosphere
33
     climate model. It assumes that the MPI implementation publishes names.
34
35
36
         MPI_Open_port(MPI_INFO_NULL, port_name);
37
         MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);
38
39
         MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
40
          /* do something with intercomm */
41
         MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);
42
43
44
     On the client side:
45
46
         MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);
47
         MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,
48
                              &intercomm);
```

Example 11.22 Simple Client-Server Example

This is a simple example; the server accepts only a single connection at a time and serves that connection until the client requests to be disconnected. The server is a single process.

Here is the server. It accepts a single connection and then processes data until it receives a message with tag 1. A message with tag 0 tells the server to exit.

```
#include "mpi.h"
                                                                                        7
int main(int argc, char *argv[])
                                                                                        8
{
                                                                                        9
    MPI_Comm client;
                                                                                        10
    MPI_Status status;
                                                                                        11
    char port_name[MPI_MAX_PORT_NAME];
                                                                                        12
    double buf[MAX_DATA];
                                                                                        13
    int
            size, again;
                                                                                        14
                                                                                        15
    MPI_Init(&argc, &argv);
                                                                                        16
    MPI_Comm_size(MPI_COMM_WORLD, &size);
                                                                                        17
    if (size != 1) error(FATAL, "Server too big");
                                                                                        18
    MPI_Open_port(MPI_INFO_NULL, port_name);
                                                                                        19
    printf("server available at %s\n", port_name);
                                                                                        20
    while (1) {
                                                                                        21
        MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
                                                                                        22
                          &client);
                                                                                        23
        again = 1;
                                                                                        ^{24}
        while (again) {
                                                                                        25
             MPI_Recv(buf, MAX_DATA, MPI_DOUBLE,
                                                                                        26
                       MPI_ANY_SOURCE, MPI_ANY_TAG, client, &status);
                                                                                        27
             switch (status.MPI_TAG) {
                                                                                        28
                case 0: MPI_Comm_free(&client);
                                                                                        29
                          MPI_Close_port(port_name);
                                                                                        30
                          MPI_Finalize();
                                                                                        31
                          return 0;
                                                                                        32
                  case 1: MPI_Comm_disconnect(&client);
                                                                                        33
                          again = 0;
                                                                                        34
                          break;
                                                                                        35
                  case 2: /* do something */
                                                                                        36
                  . . .
                                                                                        37
                  default:
                                                                                        38
                          /* Unexpected message type */
                                                                                        39
                          MPI_Abort(MPI_COMM_WORLD, 1);
                                                                                        40
                  }
                                                                                        41
             }
                                                                                        42
        }
                                                                                        43
}
                                                                                        44
                                                                                        45
Here is the client.
                                                                                        46
                                                                                        47
```

1

 $\mathbf{2}$

3

4

5 6

498 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

```
1
     #include "mpi.h"
\mathbf{2}
     int main(int argc, char **argv)
3
     ſ
4
         MPI_Comm server;
5
         double buf [MAX_DATA];
6
         char port_name[MPI_MAX_PORT_NAME];
7
8
         MPI_Init(&argc, &argv);
9
         strcpy(port_name, argv[1]);/* assume server's name is cmd-line arg */
10
11
         MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
12
                            &server);
13
14
         while (!done) {
15
              tag = 2; /* Action to perform */
16
              MPI_Send(buf, n, MPI_DOUBLE, 0, tag, server);
17
              /* etc */
18
              }
19
         MPI_Send(buf, 0, MPI_DOUBLE, 0, 1, server);
20
         MPI_Comm_disconnect(&server);
21
         MPI_Finalize();
22
         return 0;
23
     }
^{24}
25
```

11.10Other Functionality

11.10.1 Universe Size

Many "dynamic" MPI applications are expected to exist in a static runtime environment, 29 in which resources have been allocated before the application is run. When a user (or 30 possibly a batch system) runs one of these quasi-static applications, she will usually specify 31 a number of processes to start and a total number of processes that are expected. An 32 application simply needs to know how many slots there are, i.e., how many processes it 33 34 should spawn.

MPI provides an attribute on MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, that allows the 35 application to obtain this information in a portable manner. This attribute indicates the 36 total number of processes that are expected. In Fortran, the attribute is the integer value. 37 In C, the attribute is a pointer to the integer value. An application typically subtracts 38 the size of MPI_COMM_WORLD from MPI_UNIVERSE_SIZE to find out how many processes it 39 should spawn. MPI_UNIVERSE_SIZE is initialized in MPI_INIT and is not changed by MPI. If 40defined, it has the same value on all processes of MPI_COMM_WORLD. MPI_UNIVERSE_SIZE 41 is determined by the application startup mechanism in a way not specified by MPI. (The 42size of MPI_COMM_WORLD is another example of such a parameter.) 43

44

2627

28

45

47

48

46

Possibilities for how MPI_UNIVERSE_SIZE might be set include

- A -universe_size argument to a program that starts MPI processes.
- Automatic interaction with a batch scheduler to figure out how many processors have been allocated to an application.

 $\mathbf{2}$

 $\overline{7}$

- An environment variable set by the user.
- Extra information passed to MPI_COMM_SPAWN through the info argument.

An implementation must document how MPI_UNIVERSE_SIZE is set. An implementation may not support the ability to set MPI_UNIVERSE_SIZE, in which case the attribute MPI_UNIVERSE_SIZE is not set.

MPI_UNIVERSE_SIZE is a recommendation, not necessarily a hard limit. For instance, some implementations may allow an application to spawn 50 processes per processor, if they are requested. However, it is likely that the user only wants to spawn one process per processor.

MPI_UNIVERSE_SIZE is assumed to have been specified when an application was started, and is in essence a portable mechanism to allow the user to pass to the application (through the MPI process startup mechanism, such as mpiexec) a piece of critical runtime information. Note that no interaction with the runtime environment is required. If the runtime environment changes size while an application is running, MPI_UNIVERSE_SIZE is not updated, and the application must find out about the change through direct communication with the runtime system.

11.10.2 Singleton MPI Initialization

A high-quality implementation will allow any process (including those not started with a "parallel application" mechanism) to become an MPI process by calling MPI_INIT, MPI_INIT_THREAD, or MPI_SESSION_INIT. Such a process can then connect to other MPI processes using the MPI_COMM_ACCEPT and MPI_COMM_CONNECT routines, or spawn other MPI processes. MPI does not mandate this behavior, but strongly encourages it where technically feasible.

Advice to implementors. Special coordination is required to start MPI processes belonging to the same MPI_COMM_WORLD in the case of the World Model, or the same "mpi://world" process set in the Sessions Model. The processes must be started at the "same" time, they must have a mechanism to establish communication, etc. Either the user or the operating system must take special steps beyond simply starting processes.

Considering the World Model, when an application enters MPI_INIT, clearly it must be able to determine if these special steps were taken. If a process enters MPI_INIT and determines that no special steps were taken (i.e., it has not been given the information to form an MPI_COMM_WORLD with other processes) it succeeds and forms a singleton MPI program, that is, one in which MPI_COMM_WORLD has size 1.

In some implementations, MPI may not be able to function without an "MPI environment." For example, MPI may require that daemons be running or MPI may not be able to work at all on the front-end of an MPP. In this case, an MPI implementation may either

- 1. Create the environment (e.g., start a daemon) or
- 2. Raise an error if it cannot create the environment and the environment has not been started independently.

1 A high-quality implementation will try to create a singleton MPI process and not raise $\mathbf{2}$ an error. 3 (End of advice to implementors.) 4 511.10.3 MPI_APPNUM 6 $\overline{7}$ There is a predefined attribute MPI_APPNUM of MPI_COMM_WORLD. In Fortran, the at-8 tribute is an integer value. In C, the attribute is a pointer to an integer value. If a process 9 was spawned with MPI_COMM_SPAWN_MULTIPLE, MPI_APPNUM is the command number 10 that generated the current process. Numbering starts from zero. If a process was spawned 11with MPI_COMM_SPAWN, it will have MPI_APPNUM equal to zero. 12Additionally, if the process was not started by a spawn call, but by an implementation-13 specific startup mechanism that can handle multiple process specifications, MPI_APPNUM 14should be set to the number of the corresponding process specification. In particular, if it 15is started with 16mpiexec spec0 [: spec1 : spec2 : ...] 1718 MPI_APPNUM should be set to the number of the corresponding specification. 19If an application was not spawned with MPI_COMM_SPAWN or 20MPI_COMM_SPAWN_MULTIPLE, and MPI_APPNUM does not make sense in the context of 21the implementation-specific startup mechanism, MPI_APPNUM is not set. 22 MPI implementations may optionally provide a mechanism to override the value of 23MPI_APPNUM through the info argument. MPI reserves the following key for all SPAWN 24calls. 2526"appnum" Value contains an integer that overrides the default value for MPI_APPNUM in 27the child. 2829 *Rationale.* When a single application is started, it is able to figure out how many pro-30 cesses there are by looking at the size of MPI_COMM_WORLD. An application consisting 31of multiple SPMD sub-applications has no way to find out how many sub-applications 32 there are and to which sub-application the process belongs. While there are ways to 33 figure it out in special cases, there is no general mechanism. MPI_APPNUM provides 34 such a general mechanism. (End of rationale.) 35 36 **Releasing Connections** 11.10.4 37 Before a client and server connect, they are independent MPI applications. An error in one 38 does not affect the other. After establishing a connection with MPI_COMM_CONNECT and 39 MPI_COMM_ACCEPT, an error in one may affect the other. It is desirable for a client and 40 server to be able to disconnect, so that an error in one will not affect the other. Similarly, 41 it might be desirable for a parent and child to disconnect, so that errors in the child do not 42affect the parent, or vice-versa. 43 44 • Two processes are **connected** if there is a communication path (direct or indirect) 45 between them. More precisely: 46

1. Two processes are connected if

47

(a) they both belong to the same communicator (inter- or intra-, including MPI_COMM_WORLD) or	1 2
(b) they have previously belonged to a communicator that was freed with MPI_COMM_FREE instead of MPI_COMM_DISCONNECT or	3 4
(c) they both belong to the group of the same window or filehandle.	5
2. If A is connected to B and B to C, then A is connected to C.	6
2. If A is connected to b and b to C, then A is connected to C.	7
• Two processes are disconnected (also independent) if they are not connected.	8 9
• By the above definitions, connectivity is a transitive property, and divides the universe of MPI processes into disconnected (independent) sets (equivalence classes) of processes.	10 11 12
• Processes which are connected, but do not share the same MPI_COMM_WORLD, may become disconnected (independent) if the communication path between them is broken by using MPI_COMM_DISCONNECT.	13 14 15 16
The following additional rules apply to MPI routines in other chapters:	17 18
• MPI_FINALIZE is collective over a set of connected processes.	19
• MPI_ABORT does not abort independent processes. It may abort all processes in	20
the caller's MPI_COMM_WORLD (ignoring its comm argument). Additionally, it may	21
abort connected processes as well, though it makes a "best attempt" to abort only	22
the processes in comm.	23 24
• If a process terminates without calling MPI_FINALIZE, independent processes are not	25
• If a process terminates without caning MPI_FINALIZE, independent processes are not affected but the effect on connected processes is not defined.	26
anceded but the chect on connected processes is not defined.	27
Advice to implementors. In practice, it may be difficult to distinguish between an	28
MPI process failure and an erroneous program that terminates without calling an	29
MPI finalization function: an implementation that defines semantics for process fail-	30
ure management may have to exhibit the behavior defined for MPI process failures	31
with such erroneous programs. A high quality implementation should exhibit a dif- forent behavior for erroneous programs and MPI process failures. (End of advise to	32 33
ferent behavior for erroneous programs and MPI process failures. (<i>End of advice to implementors.</i>)	34
	35
	36
MDL COMM DISCONNECT (comm)	37
MPI_COMM_DISCONNECT(comm)	38
INOUT comm communicator (handle)	39
	40
C binding	41
<pre>int MPI_Comm_disconnect(MPI_Comm *comm)</pre>	42 43
Fortran 2008 binding	43 44
MPI_Comm_disconnect(comm, ierror)	45
TYPE(MPI_Comm), INTENT(INOUT) :: comm	46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	47
Fortran hinding	48

Fortran binding

Unofficial Draft for Comment Only

1 MPI_COMM_DISCONNECT(COMM, IERROR) $\mathbf{2}$

INTEGER COMM, IERROR

3

4

5

6

7

8

9

10

11

12

13 14

15

16

17

18

19 20

21

22

23 24

2526

38

39

42

43

This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation.

It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE.

MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.

Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.)

Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)

Another Way to Establish MPI Communication 11.10.5

27MPI_COMM_JOIN(fd, intercomm) 2829 IN fd socket file descriptor 30 OUT new intercommunicator (handle) intercomm 31 32 C binding 33 int MPI_Comm_join(int fd, MPI_Comm *intercomm) 34 35 Fortran 2008 binding 36 MPI_Comm_join(fd, intercomm, ierror) 37 INTEGER, INTENT(IN) :: fd

TYPE(MPI_Comm), INTENT(OUT) :: intercomm

- INTEGER, OPTIONAL, INTENT(OUT) :: ierror
- 40 Fortran binding 41

MPI_COMM_JOIN(FD, INTERCOMM, IERROR) INTEGER FD, INTERCOMM, IERROR

44MPI_COMM_JOIN is intended for MPI implementations that exist in an environment 45supporting the Berkeley Socket interface [50, 56]. Implementations that exist in an environ-46ment not supporting Berkeley Sockets should provide the entry point for MPI_COMM_JOIN 47and should return MPI_COMM_NULL. 48

This call creates an intercommunicator from the union of two MPI processes which are connected by a socket. MPI_COMM_JOIN should normally succeed if the local and remote processes have access to the same implementation-defined MPI communication universe.

Advice to users. An MPI implementation may require a specific communication medium for MPI communication, such as a shared memory segment or a special switch. In this case, it may not be possible for two processes to successfully join even if there is a socket connecting them and they are using the same MPI implementation. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will attempt to establish communication over a slow medium if its preferred one is not available. If implementations do not do this, they must document why they cannot do MPI communication over the medium used by the socket (especially if the socket is a TCP connection). (End of advice to implementors.)

fd is a file descriptor representing a socket of type SOCK_STREAM (a two-way reliable byte-stream connection). Nonblocking I/O and asynchronous notification via SIGIO must not be enabled for the socket. The socket must be in a connected state. The socket must be quiescent when MPI_COMM_JOIN is called (see below). It is the responsibility of the application to create the socket using standard socket API calls.

MPI_COMM_JOIN must be called by the process at each end of the socket. It does not return until both processes have called MPI_COMM_JOIN. The two processes are referred to as the local and remote processes.

MPI uses the socket to bootstrap creation of the intercommunicator, and for nothing else. Upon return from MPI_COMM_JOIN, the file descriptor will be open and quiescent (see below).

If MPI is unable to create an intercommunicator, but is able to leave the socket in its original state, with no pending communication, it succeeds and sets intercomm to MPI_COMM_NULL.

The socket must be quiescent before MPI_COMM_JOIN is called and after MPI_COMM_JOIN returns. More specifically, on entry to MPI_COMM_JOIN, a read on the socket will not read any data that was written to the socket before the remote process called MPI_COMM_JOIN. On exit from MPI_COMM_JOIN, a read will not read any data that was written to the socket before the remote process returned from MPI_COMM_JOIN. It is the responsibility of the application to ensure the first condition, and the responsibility of the MPI implementation to ensure the second. In a multithreaded application, the application must ensure that one thread does not access the socket while another is calling MPI_COMM_JOIN, or call MPI_COMM_JOIN concurrently.

Advice to implementors. MPI is free to use any available communication path(s) for MPI messages in the new communicator; the socket is only used for the initial handshaking. (*End of advice to implementors.*)

MPI_COMM_JOIN uses non-MPI communication to do its work. The interaction of non-MPI communication with pending MPI communication is not defined. Therefore, the result of calling MPI_COMM_JOIN on two connected processes (see Section 11.10.4 for the definition of connected) is undefined.

The returned communicator may be used to establish MPI communication with additional processes, through the usual MPI communicator creation mechanisms.

Unofficial Draft for Comment Only

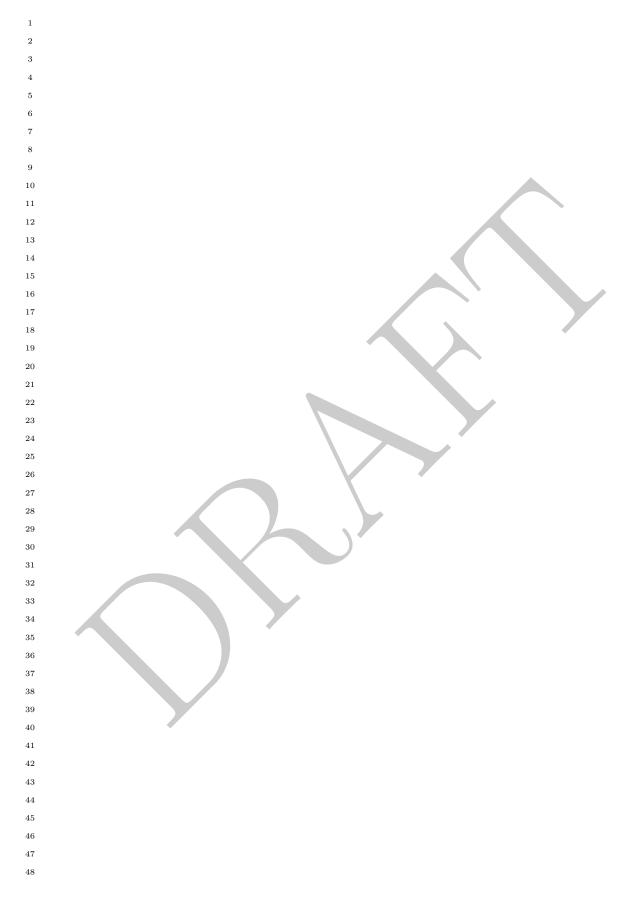
 $\mathbf{5}$

 $\overline{7}$

 24

 31

 $45 \\ 46$



Chapter 12

One-Sided Communications

12.1 Introduction

Remote Memory Access (RMA) extends the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side. This mode of communication facilitates the coding of some applications with dynamically changing data access patterns where the data distribution is fixed or slowly changing. In such a case, each process can compute what data it needs to access or to update at other processes. However, the programmer may not be able to easily determine which data in a process may need to be accessed or to be updated by operations executed by a different process, and may not even know which processes may perform such updates. Thus, the transfer parameters are all available only on one side. Regular send/receive communication requires matching operations by sender and receiver. In order to issue the matching operations, an application needs to distribute the transfer parameters. This distribution may require all processes to participate in a time-consuming global computation, or to poll for potential communication requests to receive and upon which to act periodically. The use of RMA communication mechanisms avoids the need for global computations or explicit polling. A generic example of this nature is the execution of an assignment of the form A = B(map), where map is a permutation vector, and A, B, and map are distributed in the same manner.

Message-passing communication achieves two effects: *communication* of data from sender to receiver and *synchronization* of sender with receiver. The RMA design separates these two functions. The following communication calls are provided:

- Remote write: MPI_PUT, MPI_RPUT
- Remote read: MPI_GET, MPI_RGET
- Remote update: MPI_ACCUMULATE, MPI_RACCUMULATE
- Remote read and update: MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP
- Remote atomic swap operations: MPI_COMPARE_AND_SWAP

This chapter refers to an operations set that includes all remote update, remote read and update, and remote atomic swap operations as "accumulate" operations.

9

15 16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

31

32

33

34

35

36

37 38

39 40

41

42

43 44

 $45 \\ 46$

47

1 MPI supports two fundamentally different *memory models*: separate and *unified*. The $\mathbf{2}$ separate model makes no assumption about memory consistency and is highly portable. 3 This model is similar to that of weakly coherent memory systems: the user must impose 4 correct ordering of memory accesses through synchronization calls. The unified model can $\mathbf{5}$ exploit cache-coherent hardware and hardware-accelerated, one-sided operations that are 6 commonly available in high-performance systems. The two different models are discussed 7in detail in Section 12.4. Both models support several synchronization calls to support 8 different synchronization styles.

⁹ The design of the RMA functions allows implementors to take advantage of fast or ¹⁰ asynchronous communication mechanisms provided by various platforms, such as coherent ¹¹ or noncoherent shared memory, DMA engines, hardware-supported put/get operations, and ¹² communication coprocessors. The most frequently used RMA communication mechanisms ¹³ can be layered on top of message-passing. However, certain RMA functions might need ¹⁴ support for asynchronous communication agents in software (handlers, threads, etc.) in a ¹⁵ distributed memory environment.

¹⁶ We shall denote by **origin** the process that performs the call, and by **target** the ¹⁷ process in which the memory is accessed. Thus, in a put operation, source = origin and ¹⁸ destination = target; in a get operation, source = target and destination = origin.

The use of terms such as nonblocking and local in this chapter follow the usage in
 MPI-3.1, and this chapter has not been updated to follow the definitions in Section 2.4.
 The MPI Forum intends to update this chapter in a subsequent version of the MPI standard to follow the definitions in Section 2.4.

 $23 \\ 24$

25

12.2 Initialization

²⁶ MPI provides the following window initialization functions: MPI_WIN_CREATE,

²⁷ MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and

^{2°} MPI_WIN_CREATE_DYNAMIC, which are collective on an intracommunicator.

²⁹ MPI_WIN_CREATE allows each process to specify a "window" in its memory that is made accessible to accesses by remote processes. The call returns an opaque object that represents the group of processes that own and access the set of windows, and the attributes of each window, as specified by the initialization call. MPI_WIN_ALLOCATE differs from

³⁴ MPI_WIN_CREATE in that the user does not pass allocated memory;

³⁵ MPI_WIN_ALLOCATE returns a pointer to memory allocated by the MPI implementation. ³⁵ MPI_WIN_ALLOCATE_SHARED differs from MPI_WIN_ALLOCATE in that the allocated ³⁶ memory can be accessed from all processes in the window's group with direct load/store ³⁷ instructions. Some restrictions may apply to the specified communicator.

³⁰ MPI_WIN_CREATE_DYNAMIC creates a window that allows the user to dynamically control ³⁰ which memory is exposed by the window.

- 41
- 42

- 44 45
- 46
- 47
- 48

1

12.2.1	Window Creation		1
			2
			3
MPI_W	IN_CREATE(base, size	, disp_unit, info, comm, win)	4 5
IN	base	initial address of window (choice)	6
IN	size	size of window in bytes (non-negative integer)	7
IN	disp_unit	local unit size for displacements, in bytes (positive	8
		integer)	9 10
IN	info	info argument (handle)	10
IN	comm	intra-communicator (handle)	12
OUT	win	window object (handle)	13
			14
C bind	ling		15
	•	base, MPI_Aint size, int disp_unit, MPI_Info info,	16 17
	MPI_Comm con	nm, MPI_Win *win)	18
Fortra	n 2008 binding		19
		20	
), ASYNCHRONOUS :: base	21
IN	TEGER(KIND=MPI_ADDF	ESS_KIND), INTENT(IN) :: size	22
IN	TEGER, INTENT(IN) :	: disp_unit	23
	PE(MPI_Info), INTEN		24
	PE(MPI_Comm), INTEN		25
	PE(MPI_Win), INTENT		26
IN	TEGER, OPTIONAL, IN	ITENT(OUT) :: ierror	27
Fortra	n binding		28
		C, DISP_UNIT, INFO, COMM, WIN, IERROR)	29 30
<t< td=""><td>ype> BASE(*)</td><td></td><td>31</td></t<>	ype> BASE(*)		31
IN	TEGER(KIND=MPI_ADDF	ESS_KIND) SIZE	32
IN	TEGER DISP_UNIT, IN	IFO, COMM, WIN, IERROR	33
Th	is is a collective call of	executed by all processes in the group of comm. It returns	34
		used by these processes to perform RMA operations. Each	35
		f existing memory that it exposes to RMA accesses by the	36
-		nm. The window consists of size bytes, starting at address	37
-	ŭ ·	g address of a memory region. In Fortran, one can pass the	38
		ion or a whole array, which must be 'simply contiguous' (for	39

first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see also Section 19.1.12). A process may elect to expose no memory by specifying size = 0. The displacement unit argument is provided to facilitate address arithmetic in RMA

operations: the target displacement argument of an RMA operation is scaled by the factor disp_unit specified by the target process, at window creation.

Rationale. The window size is specified using an address-sized integer, rather than a 46basic integer type, to allow windows that span more memory than can be described 47with a basic integer type. (End of rationale.) 48

Unofficial Draft for Comment Only

40

41

42

43

	308	CHAPTER 12. ONE-SIDED COMMUNICATIONS
1	Δ	<i>dvice to users.</i> Common choices for disp_unit are 1 (no scaling), and (in C syntax)
2		izeof(type), for a window that consists of an array of elements of type type. The
3		atter choice will allow one to use array indices in RMA calls, and have those scaled
4		prrectly to byte displacements, even in a heterogeneous environment. (<i>End of advice</i>
5		p users.)
6		
7	Th	e info argument provides optimization hints to the runtime about the expected usage
8	pattern	of the window. The following info keys are predefined:
9		
10		cks" — if set to true, then the implementation may assume that passive target syn-
11		nronization (i.e., MPI_WIN_LOCK, MPI_WIN_LOCK_ALL) will not be used on the
12		iven window. This implies that this window is not used for 3-party communica-
13	ti	on, and RMA can be implemented with no (less) asynchronous agent activity at this
14	p	rocess.
15	"accur	nulate_ordering" — controls the ordering of accumulate operations at the target. See
16		ection 12.7.2 for details.
17		
18	"accum	nulate_ops" — if set to "same_op", the implementation will assume that all concurrent
19	a	ccumulate calls to the same target address will use the same operation. If set to
20		same_op_no_op", then the implementation will assume that all concurrent accumulate
21		alls to the same target address will use the same operation or MPI_NO_OP. This can
22		liminate the need to protect access for certain operation types where the hardware
23	Ca	an guarantee atomicity. The default is "same_op_no_op".
24 25	"same	_size" — if set to true, then the implementation may assume that the argument size
25 26		identical on all processes, and that all processes have provided this info key with
20		ne same value.
28	01	
29		$_disp_unit"$ — if set to true, then the implementation may assume that the argument
30		isp_unit is identical on all processes, and that all processes have provided this info
31	ke	ey with the same value.
32	4	
33		<i>dvice to users.</i> The info query mechanism described in Section 12.2.7 can be used
34		o query the specified info arguments for windows that have been passed to a library. is recommended that libraries check attached info keys for each passed window.
35		End of advice to users.)
36	(1	
37	Th	e various processes in the group of comm may specify completely different target
38		rs, in location, size, displacement units, and info arguments. As long as all the get,
39		d accumulate accesses to a particular process fit their specific target window this
40	-	pose no problem. The same area in memory may appear in multiple windows, each
41		ted with a different window object. However, concurrent communications to distinct,
42	overlap	ping windows may lead to undefined results.
43	-	
44		<i>Pationale.</i> The reason for specifying the memory that may be accessed from another
45	-	rocess in an RMA operation is to permit the programmer to specify what memory
46		an be a target of RMA operations and for the implementation to enforce that spec-
47	if	ication. For example, with this definition, a server process can safely allow a client

CHAPTER 12. ONE-SIDED COMMUNICATIONS

Unofficial Draft for Comment Only

process to use RMA operations, knowing that (under the assumption that the MPI

implementation does enforce the specified limits on the exposed memory) an error in the client cannot affect any memory other than what was explicitly exposed. (End of rationale.)

A window can be created in any part of the process memory. Advice to users. However, on some systems, the performance of windows in memory allocated by MPI_ALLOC_MEM (Section 9.2) will be better. Also, on some systems, performance is improved when window boundaries are aligned at "natural" boundaries (word, double-word, cache line, page frame, etc.). (End of advice to users.)

Advice to implementors. In cases where RMA operations use different mechanisms in different memory areas (e.g., load/store in a shared memory segment, and an asynchronous handler in private memory), the MPI_WIN_CREATE call needs to figure out which type of memory is used for the window. To do so, MPI maintains, internally, the list of memory segments allocated by MPI_ALLOC_MEM, or by other, implementation-specific, mechanisms, together with information on the type of memory segment allocated. When a call to MPI_WIN_CREATE occurs, then MPI checks which segment contains each window, and decides, accordingly, which mechanism to use for RMA operations.

Vendors may provide additional, implementation-specific mechanisms to allocate or to specify memory regions that are preferable for use in one-sided communication. In particular, such mechanisms can be used to place static variables into such preferred regions.

Implementors should document any performance impact of window alignment. (End of advice to implementors.)

12.2.2 Window That Allocates Memory

MPI_WIN_	_ALLOCATE(size, disp_unit, int	fo, comm, baseptr, win)	31
IN	size	size of window in bytes (non-negative integer)	32
IN	disp_unit	local unit size for displacements, in bytes (positive	$33 \\ 34$
		integer)	35
IN	info	info argument (handle)	36
IN	comm	intra-communicator (handle)	37
OUT	baseptr	initial address of window (choice)	38
OUT	win	window object returned by call (handle)	$\frac{39}{40}$
001		window object retained by can (nandic)	40

C binding

```
int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
             MPI_Comm comm, void *baseptr, MPI_Win *win)
```

Fortran 2008 binding

```
MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
   USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
```

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9 10

11

12

13

14

15

16

17

18

19

2021

22

23

 24

25

2627

282930

42

43 44

45

46

47

1	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
2	INTEGER, INTENT(IN) :: disp_unit
3	TYPE(MPI_Info), INTENT(IN) :: info
4	TYPE(MPI_Comm), INTENT(IN) :: comm
5	TYPE(C_PTR), INTENT(OUT) :: baseptr
6	TYPE(MPI_Win), INTENT(OUT) :: win
7	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8	INTEGER, OFITOWAE, INTENT(001) TETTOT
9	Fortran binding
10	MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
11	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
12	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
13	This is a collective call executed by all processes in the group of comm. On each
14	process, it allocates memory of at least size bytes, returns a pointer to it, and returns a
15	window object that can be used by all processes in comm to perform RMA operations. The
16	returned memory consists of size bytes local to each process, starting at address baseptr
17	and is associated with the window as if the user called MPI_WIN_CREATE on existing
18	memory. The size argument may be different at each process and size = 0 is valid; however, a
19	library might allocate and expose more memory in order to create a fast, globally symmetric
20	allocation. The discussion of and rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in
21	Section 9.2 also apply to MPI_WIN_ALLOCATE; in particular, see the rationale in Section 9.2
22	for an explanation of the type used for baseptr .
23	If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must
24	be provided in the mpi module and should be provided in mpif.h through overloading,
25	i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
26	BASEPTR, but with a different specific procedure name:
27	
28	INTERFACE MPI_WIN_ALLOCATE
29	SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
30	WIN, IERROR)
31	IMPORT :: MPI_ADDRESS_KIND
32	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
33	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
34	END SUBROUTINE
35	SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
36	WIN, IERROR)
37	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
38	IMPORT :: MPI_ADDRESS_KIND
39	INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
40	INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
41	TYPE(C_PTR) :: BASEPTR
42	END SUBROUTINE
43	END INTERFACE
44	The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_CPTR.
45	The implied specific procedure names are described in Section 19.1.5.
46	
47	Rationale. By allocating (potentially aligned) memory instead of allowing the user
48	to pass in an arbitrary buffer, this call can improve the performance for systems with

Unofficial Draft for Comment Only

remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (End of rationale.)

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM.

The default memory alignment requirements and the "mpi_minimum_memory_alignment" info key described for MPI_ALLOC_MEM in Section 9.2 apply to all processes with non-zero size argument.

12.2.3 Window That Allocates Shared Memory

MPI_WIN_ALLOCATE_SHARED(size, disp_unit, info, comm, baseptr, win)

-			16
IN	size	size of local window in bytes (non-negative integer)	17
IN	disp_unit	local unit size for displacements, in bytes (positive	18
		integer)	19
IN	info	info argument (handle)	20
			21
IN	comm	intra-communicator (handle)	22
OU	T baseptr	address of local allocated window segment (choice)	23
ΟU	T win	window object returned by the call (handle)	24
			25

C binding

int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win)

Fortran 2008 binding

MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)	31
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	32
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size	33
INTEGER, INTENT(IN) :: disp_unit	34
TYPE(MPI_Info), INTENT(IN) :: info	35
TYPE(MPI_Comm), INTENT(IN) :: comm	36
TYPE(C_PTR), INTENT(OUT) :: baseptr	37
TYPE(MPI_Win), INTENT(OUT) :: win	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
Fortron hinding	40
Fortran binding	41
MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)	42

INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR

This is a collective call executed by all processes in the group of comm. On each 4546process, it allocates memory of at least size bytes that is shared among all processes in 47comm, and returns a pointer to the locally allocated segment in **baseptr** that can be used 48 for load/store accesses on the calling process. The locally allocated memory can be the

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

7

8

9

10 11

26

27

28

29

30

43

1 target of load/store accesses by remote processes; the base pointers for other processes $\mathbf{2}$ can be queried using the function MPI_WIN_SHARED_QUERY. The call also returns a 3 window object that can be used by all processes in comm to perform RMA operations. 4 The size argument may be different at each process and size = 0 is valid. It is the user's $\mathbf{5}$ responsibility to ensure that the communicator comm represents a group of processes that 6 can create a shared memory segment that can be accessed by all processes in the group. 7The discussions of rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in Section 9.2 8 also apply to MPI_WIN_ALLOCATE_SHARED; in particular, see the rationale in Section 9.2 9 for an explanation of the type used for **baseptr**. The allocated memory is contiguous across 10 process ranks unless the info key "alloc_shared_noncontig" is specified. Contiguous across 11process ranks means that the first address in the memory segment of process i is consecutive 12with the last address in the memory segment of process i-1. This may enable the user to 13calculate remote address offsets with local information only. 14If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must 15be provided in the mpi module and should be provided in mpif.h through overloading, 16i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND) 17BASEPTR, but with a different specific procedure name: 18

```
<sup>19</sup> INTERFACE MPI_WIN_ALLOCATE_SHARED
```

users.)

20	SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
21	BASEPTR, WIN, IERROR)
22	IMPORT :: MPI_ADDRESS_KIND
23	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
24	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
25	END SUBROUTINE
26	SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
27	BASEPTR, WIN, IERROR)
28	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
29	IMPORT :: MPI_ADDRESS_KIND
30	INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
31	INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
32	TYPE(C_PTR) :: BASEPTR
33	END SUBROUTINE
34	END INTERFACE
35	
36	The base procedure name of this overloaded function is
37	MPI_WIN_ALLOCATE_SHARED_CPTR. The implied specific procedure names are described
38	in Section 19.1.5.
39	The info argument can be used to specify hints similar to the info argument for
40	MPI_WIN_CREATE, MPI_WIN_ALLOCATE, and MPI_ALLOC_MEM. The additional info
41	key "alloc_shared_noncontig" allows the library to optimize the layout of the shared memory
42	segments in memory.
43	
44	Advice to users. If the info key "alloc_shared_noncontig" is not set to true, the allocation
45	strategy is to allocate contiguous memory across process ranks. This may limit the
46	performance on some architectures because it does not allow the implementation to
47	modify the data layout (e.g., padding to reduce access latency). (End of advice to

Advice to implementors. If the user sets the info key "alloc_shared_noncontig" to true, the implementation can allocate the memory requested by each process in a location that is close to this process. This can be achieved by padding or allocating memory in special memory segments. Both techniques may make the address space across consecutive ranks noncontiguous. (End of advice to implementors.)

For contiguous shared memory allocations, the default alignment requirements outlined for MPI_ALLOC_MEM in Section 9.2 and the "mpi_minimum_memory_alignment" info key apply to the start of the contiguous memory that is returned in baseptr to the first process with non-zero size argument. For noncontiguous memory allocations, the default alignment requirements and the "mpi_minimum_memory_alignment" info key apply to all processes with non-zero size argument.

Advice to users. If the info key "alloc_shared_noncontig" is not set to true (or ignored by the MPI implementation), the alignment of the memory returned in baseptr to all but the first process with non-zero size argument depends on the value of the size argument provided by other processes. It is thus the user's responsibility to control the alignment of contiguous memory allocated for these processes by ensuring that each process provides a size argument that is an integral multiple of the alignment required for the application. (End of advice to users.)

The consistency of load/store accesses from/to the shared memory as observed by the user program depends on the architecture. A consistent view can be created in the *unified memory model* (see Section 12.4) by utilizing the window synchronization functions (see Section 12.5) or explicitly completing outstanding store accesses (e.g., by calling MPI_WIN_FLUSH). MPI does not define semantics for accessing shared memory windows in the *separate memory model*.

	SHARED UTERYIMIN FANK 9	SIZA AIGN IINIT NOSANTRI	
	SHARED_QUERY (win, rank, s	size, disp_difit, basepti)	30
IN	win	shared memory window object (handle)	31
IN	rank	rank in the group of window win or	32
		MPI_PROC_NULL (non-negative integer)	33
OUT	size	size of the window segment (non-negative integer)	34
	dian unit		35
OUT	disp_unit	local unit size for displacements, in bytes (positive	36
		integer)	37
OUT	baseptr	address for load/store access to window segment	38
		(choice)	39
			40
C bindin	σ		41
	0	vin, int rank, MPI_Aint *size,	42
	int *disp_unit, void		43
	1110 albp_a110, vola		44
Fortran 2	2008 binding		45
MPI_Win_s	shared_query(win, rank, si	ize, disp_unit, baseptr, ierror)	46
USE,	INTRINSIC :: ISO_C_BINDIN	NG, ONLY : C_PTR	47

MPI_WIN_SHARED_QUERY(win, rank, size, disp_unit, baseptr)

TYPE(MPI_Win), INTENT(IN) :: win

```
Unofficial Draft for Comment Only
```

1

 $\mathbf{2}$

3

4

5

6 7

8

9

10

11

12 13

14

15

16

17

18

19

20 21

22

23

24

25

26

27 28 29

1	INTEGER, INTENT(IN) :: rank
2	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
3	INTEGER, INTENT(OUT) :: disp_unit
4	TYPE(C_PTR), INTENT(OUT) :: baseptr
5	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6	
7	Fortran binding
8	MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)
9	INTEGER WIN, RANK, DISP_UNIT, IERROR
10	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
11	This function queries the process-local address for remote memory segments created
12	with MPI_WIN_ALLOCATE_SHARED. This function can return different process-local ad-
13	dresses for the same physical memory on different processes. The returned memory can be
14	used for load/store accesses subject to the constraints defined in Section 12.7. This function
15	can only be called with windows of flavor MPI_WIN_FLAVOR_SHARED. If the passed window
16	is not of flavor MPI_WIN_FLAVOR_SHARED, the error MPI_ERR_RMA_FLAVOR is raised. When
17	rank is MPI_PROC_NULL, the pointer, disp_unit, and size returned are the pointer, disp_unit,
18	and size of the memory segment belonging the lowest rank that specified size > 0 . If all
19	processes in the group attached to the window specified $size = 0$, then the call returns $size$
20	= 0 and a baseptr as if MPI_ALLOC_MEM was called with size = 0.
21	If the Fortran compiler provides $\texttt{TYPE}(\texttt{C}_\texttt{PTR})$, then the following generic interface must
22	be provided in the mpi module and should be provided in mpif.h through overloading,
23	i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
24 25	BASEPTR, but with a different specific procedure name:
25 26	INTERFACE MPI_WIN_SHARED_QUERY
27	SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &
28	BASEPTR, IERROR)
29	IMPORT :: MPI_ADDRESS_KIND
30	INTEGER WIN, RANK, DISP_UNIT, IERROR
31	INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
32	END SUBROUTINE
33	
33 34	SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR)
	SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &
34	SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR)
34 35	SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
34 35 36	SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND
34 35 36 37	SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: WIN, RANK, DISP_UNIT, IERROR
34 35 36 37 38	<pre>SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR END SUBROUTINE</pre>
34 35 36 37 38 39	SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR
34 35 36 37 38 39 40 41 42	<pre>SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR END SUBROUTINE END INTERFACE</pre>
34 35 36 37 38 39 40 41 42 43	SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR END SUBROUTINE END INTERFACE The base procedure name of this overloaded function is
34 35 36 37 38 39 40 41 42 43 44	SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR END SUBROUTINE END INTERFACE The base procedure name of this overloaded function is MPI_WIN_SHARED_QUERY_CPTR. The implied specific procedure names are described in
34 35 36 37 38 39 40 41 42 43 44 45	SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR END SUBROUTINE END INTERFACE The base procedure name of this overloaded function is
34 35 36 37 38 39 40 41 42 43 44 45 46	SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR END SUBROUTINE END INTERFACE The base procedure name of this overloaded function is MPI_WIN_SHARED_QUERY_CPTR. The implied specific procedure names are described in
34 35 36 37 38 39 40 41 42 43 44 45	SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR END SUBROUTINE END INTERFACE The base procedure name of this overloaded function is MPI_WIN_SHARED_QUERY_CPTR. The implied specific procedure names are described in

12.2.4 Window of Dynamically Attached Memory

The MPI-2 RMA model requires the user to identify the local memory that may be a target of RMA calls at the time the window is created. This has advantages for both the programmer (only this memory can be updated by one-sided operations and provides greater safety) and the MPI implementation (special steps may be taken to make one-sided access to such memory more efficient). However, consider implementing a modifiable linked list using RMA operations; as new items are added to the list, memory must be allocated. In a C or C++ program, this memory is typically allocated using malloc or new respectively. In MPI-2 RMA, the programmer must create a window with a predefined amount of memory and then implement routines for allocating memory from within the window's memory. In addition, there is no easy way to handle the situation where the predefined amount of memory turns out to be inadequate. To support this model, the routine MPI_WIN_CREATE_DYNAMIC creates a window that makes it possible to expose memory without remote synchronization. It must be used in combination with the local routines MPI_WIN_ATTACH and MPI_WIN_DETACH.

MPI_WIN_CREATE_DYNAMIC(info, comm, win)

IN	info	info argument (handle)
IN	comm	intra-communicator (handle)
OUT	win	window object returned by the call (handle)

C binding

int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)
Fortran 2008 binding
MPI_Win_create_dynamic(info, comm, win, ierror)
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Win), INTENT(OUT) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran hinding

Fortran binding

```
MP1_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)
INTEGER INFO, COMM, WIN, IERROR
```

This is a collective call executed by all processes in the group of comm. It returns a window win without memory attached. Existing process memory can be attached as described below. This routine returns a window object that can be used by these processes to perform RMA operations on attached memory. Because this window has special properties, it will sometimes be referred to as a *dynamic* window.

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE.

In the case of a window created with MPI_WIN_CREATE_DYNAMIC, the target_disp for all RMA functions is the address at the target; i.e., the effective window_base is MPI_BOTTOM and the disp_unit is one. For dynamic windows, the target_disp argument to RMA communication operations is not restricted to non-negative values. Users should use

Unofficial Draft for Comment Only

 $\mathbf{2}$

	516	C_{\perp}	HAPTER 12. ONE-SIDED COMMUNICATIONS
1 2 3		_ADDRESS at the target pro- and communicate this address	cess to determine the address of a target memory to the origin process.
4 5 6 7	vari MP	iables of type MPI_Aint and res I_AINT_ADD and MPI_AINT_I	ioned that displacement arithmetic can overflow in sult in unexpected values on some platforms. The DIFF functions can be used to safely perform address ments. (<i>End of advice to users.</i>)
8 9 10 11 12 13 14 15	care it is can poin typ	e must be exercised in commun s possible that an address valid not be expressed as an address nters). For this reason, a por	ronments with heterogeneous data representations, nicating addresses between processes. For example, at the target process (for example, a 64-bit pointer) is at the origin (for example, the origin uses 32-bit table MPI implementation should ensure that the able to store addresses from any process. (<i>End of</i>
16 17 18 19 20 21	been atta MPI_WIN MPI_WIN	ached using the function $MPI_{}^{}$	accessed with this window until that memory has WIN_ATTACH. That is, in addition to using te an MPI window, the user must use emory may be the target of an MPI RMA operation. le may be attached.
22			
23 24	MPI_WIN	I_ATTACH(win, base, size)	
24	IN	win	window object (handle)
26	IN	base	initial address of memory to be attached (choice)
27 28 29	IN	size	size of memory to be attached in bytes (non-negative integer)
30			
31	C bindi	0	
32	int MPI_	_Win_attach(MPI_Win win, v	oid *base, MPI_Aint size)
33	Fortran	2008 binding	
34		_attach(win, base, size, i	
35		E(MPI_Win), INTENT(IN) ::	
36 37		E(*), DIMENSION(), ASYNC	
38		EGER(KIND=MPI_ADDRESS_KIND EGER, OPTIONAL, INTENT(OUT	-
39) ieiioi
40	Fortran		
41		ATTACH(WIN, BASE, SIZE, I	ERROR)
42		EGER WIN, IERROR De> BASE(*)	
43	• -	EGER(KIND=MPI_ADDRESS_KIND) SIZE
44			
45			eginning at base for remote access within the given
46 47			must not contain any part that is already attached
48		, , , ,	overlapping memory concurrently within the same in must be a window that was created with

MPI_WIN_CREATE_DYNAMIC. The local memory region attached to the window consists of size bytes, starting at address base. In C, base is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see Section 19.1.12). Multiple (but non-overlapping) memory regions may be attached to the same window.

Rationale. Requiring that memory be explicitly attached before it is exposed to one-sided access by other processes can simplify implementations and improve performance. The ability to make memory available for RMA operations without requiring a collective MPI_WIN_CREATE call is needed for some one-sided programming models. (*End of rationale.*)

Advice to users. Attaching memory to a window may require the use of scarce resources; thus, attaching large regions of memory is not recommended in portable programs. Attaching memory to a window may fail if sufficient resources are not available; this is similar to the behavior of MPI_ALLOC_MEM.

The user is also responsible for ensuring that MPI_WIN_ATTACH at the target has returned before a process attempts to target that memory with an MPI RMA call.

Performing an RMA operation to memory that has not been attached to a window created with MPI_WIN_CREATE_DYNAMIC is erroneous. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will attempt to make as much memory available for attaching as possible. Any limitations should be documented by the implementor. (*End of advice to implementors.*)

Attaching memory is a local operation as defined by MPI, which means that the call is not collective and completes without requiring any MPI routine to be called in any other process. Memory may be detached with the routine MPI_WIN_DETACH. After memory has been detached, it may not be the target of an MPI RMA operation on that window (unless the memory is re-attached with MPI_WIN_ATTACH).

		02
MPI_WIN_DETACH(win, base)		33
IN win	window object (handle)	34
		35
IN base	initial address of memory to be detached (choice)	36
		37
C binding		38
<pre>int MPI_Win_detach(MPI_Win win, co</pre>	onst void *base)	39
Fortron 2008 binding		40
Fortran 2008 binding		41
<pre>MPI_Win_detach(win, base, ierror)</pre>		42
TYPE(MPI_Win), INTENT(IN) :: w	vin	
TYPE(*), DIMENSION(), ASYNCH	RONOUS :: base	43
INTEGER, OPTIONAL, INTENT(OUT)		44
INTEGER, OFTIONAL, INTENT(001)	161101	45
Fortran binding		46
MPI_WIN_DETACH(WIN, BASE, IERROR)		47
INTEGER WIN, IERROR		48

 24

1	<type> BASE(*)</type>
2 3 4	Detaches a previously attached memory region beginning at base. The arguments base and win must match the arguments passed to a previous call to MPI_WIN_ATTACH.
5 6 7 8 9	Advice to users. Detaching memory may permit the implementation to make more efficient use of special memory or provide memory that may be needed by a subsequent MPI_WIN_ATTACH. Users are encouraged to detach memory that is no longer needed. Memory should be detached before it is freed by the user. (<i>End of advice to users.</i>)
10 11 12	Memory becomes detached when the associated dynamic memory window is freed, see Section 12.2.5.
13 14 15	12.2.5 Window Destruction
16 17	MPI_WIN_FREE(win)
17	INOUT win window object (handle)
19	
20 21	C binding
22	<pre>int MPI_Win_free(MPI_Win *win)</pre>
23 24 25 26	Fortran 2008 binding MPI_Win_free(win, ierror) TYPE(MPI_Win), INTENT(INOUT) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27 28 29 30	Fortran binding MPI_WIN_FREE(WIN, IERROR) INTEGER WIN, IERROR
 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 	Frees the window object win and returns a null handle (equal to MPI_WIN_NULL). This is a collective call executed by all processes in the group associated with win. MPI_WIN_FREE(win) can be invoked by a process only after it has completed its involvement in RMA communications on window win: e.g., the process has called MPI_WIN_FENCE, or called MPI_WIN_WAIT to match a previous call to MPI_WIN_POST or called MPI_WIN_COMPLETE to match a previous call to MPI_WIN_START or called MPI_WIN_UNLOCK to match a previous call to MPI_WIN_START or called MPI_WIN_UNLOCK to match a previous call to MPI_WIN_LOCK. The memory associated with windows created by a call to MPI_WIN_CREATE may be freed after the call returns. If the window was created with MPI_WIN_ALLOCATE, MPI_WIN_FREE will free the window memory that was allocated in MPI_WIN_ALLOCATE. If the window was created with MPI_WIN_ALLOCATE_SHARED, MPI_WIN_FREE will free the window memory that was allocated in MPI_WIN_ALLOCATE.SHARED. Freeing a window that was created with a call to MPI_WIN_CREATE_DYNAMIC de- taches all associated memory; i.e., it has the same effect as if all attached memory was detached by calls to MPI_WIN_DETACH. Advice to implementors. MPI_WIN_FREE requires a barrier synchronization: no pro-
47 48	cess can return from free until all processes in the group of win call free. This ensures

that no process will attempt to access a remote window (e.g., with lock/unlock) after it was freed. The only exception to this rule is when the user sets the "no_locks" info key to "true" when creating the window. In that case, an MPI implementation may free the local window without barrier synchronization. (*End of advice to implementors.*)

12.2.6 Window Attributes

The following attributes are cached with a window when the window is created.

MPI_WIN_BASE	window base address.
MPI_WIN_SIZE	window size, in bytes.
MPI_WIN_DISP_UNIT	displacement unit associated with the window.
MPI_WIN_CREATE_FLAVOR	how the window was created.
MPI_WIN_MODEL	memory model for window.

In C, calls to MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag), MPI_Win_get_attr(win, MPI_WIN_SIZE, &size, &flag), MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag), MPI_Win_set_attr(win, MPI_WIN_CDEATE_EIA)(OD_and to be displayed)

MPI_Win_get_attr(win, MPI_WIN_CREATE_FLAVOR, &create_kind, &flag), and MPI_Win_get_attr(win, MPI_WIN_MODEL, &memory_model, &flag) will return in base a pointer to the start of the window win, and will return in size, disp_unit, create_kind, and memory_model pointers to the size, displacement unit of the window, the kind of routine used to create the window, and the memory model, respectively. A detailed listing of the type of the pointer in the attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR is shown in Table 12.1.

Attribute	V	СЛ	ype
MPI_WIN_BASE		void	1 *
MPI_WIN_SIZE		MPI.	_Aint *
MPI_WIN_DISP_UN	JIT TIV	int	*
MPI_WIN_CREATE	_FLAVOR	int	*
MPI_WIN_MODEL		int	*

Table 12.1: C types of attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR.

In Fortran, calls to MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, base, flag, ierror), MPI_WIN_GET_ATTR(win, MPI_WIN_SIZE, size, flag, ierror), MPI_WIN_GET_ATTR(win, MPI_WIN_DISP_UNIT, disp_unit, flag, ierror), MPI_WIN_GET_ATTR(win, MPI_WIN_CREATE_FLAVOR, create_kind, flag, ierror), and MPI_WIN_GET_ATTR(win, MPI_WIN_MODEL, memory_model, flag, ierror) will return in base, size, disp_unit, create_kind, and memory_model the (integer representation of) the base address, the size, the displacement unit of the window win, the kind of routine used to create the window, and the memory model, respectively.

The values of create_kind are

MPI_WIN_FLAVOR_CREATE	Window was created with MPI_WIN_CREATE.	46
MPI_WIN_FLAVOR_ALLOCATE	Window was created with MPI_WIN_ALLOCATE.	47
		48

1

 $\mathbf{2}$

3

4 5 6

7

15

16

17

18

19

20

21

22

23

34

35 36

37

38

39

40

41

42

43

44

```
1
       MPI_WIN_FLAVOR_DYNAMIC
                                             Window was created with
2
                                             MPI_WIN_CREATE_DYNAMIC.
3
                                             Window was created with
       MPI_WIN_FLAVOR_SHARED
4
                                             MPI_WIN_ALLOCATE_SHARED.
5
          The values of memory_model are MPI_WIN_SEPARATE and MPI_WIN_UNIFIED. The mean-
6
     ing of these is described in Section 12.4.
7
         In the case of windows created with MPI_WIN_CREATE_DYNAMIC, the base address
8
     is MPI_BOTTOM and the size is 0. In C, pointers are returned, and in Fortran, the values are
9
     returned, for the respective attributes. (The window attribute access functions are defined
10
     in Section 7.7.3.) The value returned for an attribute on a window is constant over the
11
     lifetime of the window.
12
          The other "window attribute," namely the group of processes attached to the window,
13
     can be retrieved using the call below.
14
15
16
     MPI_WIN_GET_GROUP(win, group)
17
       IN
                                             window object (handle)
                 win
18
       OUT
                                             group of processes which share access to the window
                 group
19
                                             (handle)
20
21
22
     C binding
23
     int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
^{24}
     Fortran 2008 binding
25
     MPI_Win_get_group(win, group, ierror)
26
          TYPE(MPI_Win), INTENT(IN) :: win
27
          TYPE(MPI_Group), INTENT(OUT) :: group
28
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     Fortran binding
^{31}
     MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
32
          INTEGER WIN, GROUP, IERROR
33
          MPI_WIN_GET_GROUP returns a duplicate of the group of the communicator used to
34
     create the window associated with win. The group is returned in group.
35
36
            Window Info
     12.2.7
37
```

CHAPTER 12. ONE-SIDED COMMUNICATIONS

38 Hints specified via info (see Section 10) allow a user to provide information to direct opti-39 mization. Providing hints may enable an implementation to deliver increased performance 40 or use system resources more efficiently. An implementation is free to ignore all hints; 41 however, applications must comply with any info hints they provide that are used by the 42MPI implementation (i.e., are returned by a call to MPI_WIN_GET_INFO) and that place 43 a restriction on the behavior of the application. Hints are specified on a per window basis, 44 in window creation functions and MPI_WIN_SET_INFO, via the opaque info object. When 45an info object that specifies a subset of valid hints is passed to MPI_WIN_SET_INFO there 46will be no effect on previously set or default hints that the info does not specify. 47

48

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for the hint. (*End of advice to implementors.*)

			8
MPI_WIN	_SET_INFO(wi	n, info)	9
- INOUT	win	window object (handle)	10 11
			12
IN	info	info argument (handle)	13
α \cdot \cdot \cdot			14
C bindin	0	(MDT lin min MDT Info info)	15
INC MPI_	win_set_into	(MPI_Win win, MPI_Info info)	16
	2008 binding		17
		info, ierror)	18
	-	VTENT(IN) :: win	19
	-	INTENT(IN) :: info	20
INTE	GER, UPTIONAL	., INTENT(OUT) :: ierror	21
Fortran	binding		22 23
MPI_WIN_	SET_INFO(WIN,	, INFO, IERROR)	23 24
INTE	GER WIN, INFO), IERROR	25
MPI	WIN SET INF	O updates the hints of the window associated with win using the	26
		This operation has no effect on previously set or defaulted hints	27
-		info. It also has no effect on previously set or defaulted hints that	28
are specif	ied by i nfo , bu	t are ignored by the MPI implementation in this call to	29
MPI_WIN	_SET_INFO. T	he call is collective on the group of win. The info object may be	30
	-	s, but any info entries that an implementation requires to be the	31
same on a	all processes mu	ist appear with the same value in each process's info object.	32
Ada	ice to users.	Some info items that an implementation can use when it creates	33
		easily be changed once the window has been created. Thus, an	34 35
		y ignore hints issued in this call that it would have accepted in a	36
-		aplementation may also be unable to update certain info hints in a	37
call	to MPI_WIN_S	ET_INFO. MPI_WIN_GET_INFO can be used to determine whether	38
info	changes were i	gnored by the implementation. (End of advice to users.)	39
			40
			41
MPI_WIN	_GET_INFO(wi	n, info_used)	42
IN	win	window object (handle)	43
			44
OUT	info_used	new info object (handle)	45
			46
C bindin	•		47
int MPf_	Win_get_info((MPI_Win win, MPI_Info *info_used)	48

int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)

1	Fortran 2008 binding
2	MPI_Win_get_info(win, info_used, ierror)
3	TYPE(MPI_Win), INTENT(IN) :: win
4	TYPE(MPI_Info), INTENT(OUT) :: info_used
5	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6	
7	Fortran binding
	MOT LIN CET INCOLUIN INCO USED IEDDOD)

MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR) INTEGER WIN, INFO_USED, IERROR

MPI_WIN_GET_INFO returns a new info object containing the hints of the window associated with win. The current setting of all hints related to this window is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info_used via MPI_INFO_FREE.

¹⁰ 12.3 Communication Calls

²⁰ MPI supports the following RMA communication calls: MPI_PUT and MPI_RPUT transfer ²¹ data from the caller memory (origin) to the target memory; MPI_GET and MPI_RGET ²³ transfer data from the target memory to the caller memory; MPI_ACCUMULATE and ²⁴ MPI_RACCUMULATE update locations in the target memory, e.g., by adding to these lo-²⁵ cations values sent from the caller memory; MPI_GET_ACCUMULATE,

MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP perform atomic read-modify-write 26and return the data before the accumulate operation; and MPI_COMPARE_AND_SWAP per-27forms a remote atomic compare and swap operation. These operations are *nonblocking*: the 28 call initiates the transfer, but the transfer may continue after the call returns. The transfer 29 is completed, at the origin or both the origin and the target, when a subsequent synchro-30 nization call is issued by the caller on the involved window object. These synchronization 31 calls are described in Section 12.5. Transfers can also be completed with calls to flush rou-32 tines; see Section 12.5.4 for details. For the MPI_RPUT, MPI_RGET, MPI_RACCUMULATE, 33 and MPI_RGET_ACCUMULATE calls, the transfer can be locally completed by using the 34MPI test or wait operations described in Section 3.7.3. 35

The local communication buffer of an RMA call should not be updated, and the local communication buffer of a get call should not be accessed after the RMA call until the operation completes at the origin.

The resulting data values, or outcome, of concurrent conflicting accesses to the same 39 memory locations is undefined; if a location is updated by a put or accumulate operation, 40 then the outcome of loads or other RMA operations is undefined until the updating operation 41 has completed at the target. There is one exception to this rule; namely, the same location 42can be updated by several concurrent accumulate calls, the outcome being as if these updates 43 occurred in some order. In addition, the outcome of concurrent load/store and RMA updates 44to the same memory location is undefined. These restrictions are described in more detail 45in Section 12.7. 46

The calls use general datatype arguments to specify communication buffers at the origin and at the target. Thus, a transfer operation may also gather data at the source and scatter

8

9 10

11

12

13

14

15

16

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8 9

10

11

12

13 14

1516

17

18

1920

it at the destination. However, all arguments specifying both communication buffers are provided by the caller.

For all RMA calls, the target process may be identical with the origin process; i.e., a process may use an RMA operation to move data in its memory.

Rationale. The choice of supporting "self-communication" is the same as for messagepassing. It simplifies some coding, and is very useful with accumulate operations, to allow atomic updates of local variables. (End of rationale.)

MPI_PROC_NULL is a valid target rank in all MPI RMA communication calls. The effect is the same as for MPI_PROC_NULL in MPI point-to-point communication. After any RMA operation with rank MPI_PROC_NULL, it is still necessary to finish the RMA epoch with the synchronization method that started the epoch.

12.3.1 Put

The execution of a put operation is similar to the execution of a send by the origin process and a matching receive by the target process. The obvious difference is that all arguments are provided by one call — the call executed by the origin process.

MPI_PU	Г(origin_addr, origin_count, orig	in_datatype, target_rank, target_disp, target_count,	21
	target_datatype, win)		22
IN	origin_addr	initial address of origin buffer (choice)	23 24
IN	origin_count	number of entries in origin buffer (non-negative	24 25
IIV	oligin_count	integer)	26
IN	origin_datatype	datatype of each entry in origin buffer (handle)	27
IN	target_rank	rank of target (non-negative integer)	28
			29
IN	target_disp	displacement from start of window to target buffer	30
		(non-negative integer)	31
IN	target_count	number of entries in target buffer (non-negative	32
		integer)	33
IN	target_datatype	datatype of each entry in target buffer (handle)	34
		• • • • • • •	35
IN	win	window object used for communication (handle)	36
			37
C bindi	ng		38
int MPI	_Put(const void *origin_ad	dr, int origin_count,	39
	MPI_Datatype origin_	_datatype, int target_rank,	40
	MPI_Aint target_disp	o, int target_count,	41
	MPI_Datatype target_	_datatype, MPI_Win win)	42
			43
	2008 binding		44
MPI_Put	0	, origin_datatype, target_rank,	45
יריזים	o i o	count, target_datatype, win, ierror)	46
		T(IN), ASYNCHRONOUS :: origin_addr	47
TNU	LGER, INTENT(IN) :: origin	_count, target_rank, target_count	48

1	TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
3	TYPE(MPI_Win), INTENT(IN) :: win
4	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5	INTEGER, OFFICIARE, INTENT(OOF) TETTOT
6	Fortran binding
7	MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
8	TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
9	<type> ORIGIN_ADDR(*)</type>
10	INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
11	TARGET_DATATYPE, WIN, IERROR
12	INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
13	Transfers origin_count successive entries of the type specified by the origin_datatype,
14	starting at address origin_addr on the origin node, to the target node specified by the win,
15	target_rank pair. The data are written in the target buffer at address target_addr =
16	window_base+target_disp × disp_unit, where window_base and disp_unit are the base address
17	and window displacement unit specified at window initialization, by the target process.
18	The target buffer is specified by the arguments target_count and target_datatype.
19	The data transfer is the same as that which would occur if the origin process executed
20	a send operation with arguments origin_addr, origin_count, origin_datatype, target_rank, tag,
21	comm, and the target process executed a receive operation with arguments target_addr,
22	target_count, target_datatype, source, tag, comm, where target_addr is the target buffer
23	address computed as explained above, the values of tag are arbitrary valid matching tag
24	values, and comm is a communicator for the group of win.
25	The communication must satisfy the same constraints as for a similar message-passing
26	communication. The target_datatype may not specify overlapping entries in the target
27	buffer. The message sent must fit, without truncation, in the target buffer. Furthermore,
28	the target buffer must fit in the target window or in attached memory in a dynamic window.
29	The target_datatype argument is a handle to a datatype object defined at the origin
30	process. However, this object is interpreted at the target process: the outcome is as if
31	the target datatype object was defined at the target process by the same sequence of calls
32	used to define it at the origin process. The target datatype must contain only relative
33	displacements, not absolute addresses. The same holds for get and accumulate operations.
34	
35	Advice to users. The target_datatype argument is a handle to a datatype object that
36	is defined at the origin process, even though it defines a data layout in the target
37	process memory. This causes no problems in a homogeneous environment, or in a
38	heterogeneous environment if only portable datatypes are used (portable datatypes
39 40	are defined in Section 2.4).
40	The performance of a put transfer can be significantly affected, on some systems, by
42	the choice of window location and the shape and location of the origin and target
42	buffer: transfers to a target window in memory allocated by MPI_ALLOC_MEM or
44	MPI_WIN_ALLOCATE may be much faster on shared memory systems; transfers from
45	contiguous buffers will be faster on most, if not all, systems; the alignment of the
46	communication buffers may also impact performance. (End of advice to users.)
47	Advice to implementors. A high-quality implementation will attempt to prevent
48	remote accesses to memory outside the window that was exposed by the process.
	Tomote accesses to memory outside the window that was exposed by the process.

Unofficial Draft for Comment Only

This is important both for debugging purposes and for protection with client-server codes that use RMA. That is, a high-quality implementation will check, if possible, window bounds on each RMA call, and raise an error at the origin call if an out-of-bound situation occurs. Note that the condition can be checked at the origin. Of course, the added safety achieved by such checks has to be weighed against the added cost of such checks. (*End of advice to implementors.*)

12.3.2 Get

MPI GE	T(origin addr. origin count. (origin_datatype, target_rank, target_disp, target_count,	11
_	target_datatype, win)		12
OUT	origin_addr	initial address of origin buffer (choice)	13 14
IN	origin_count	number of entries in origin buffer (non-negative	15
	ongm_count	integer)	16
IN	origin_datatype	datatype of each entry in origin buffer (handle)	17
IN	target_rank	rank of target (non-negative integer)	18
IN	target_disp	displacement from window start to the beginning of	19 20
IIN	target_uisp	the target buffer (non-negative integer)	20 21
IN	target_count	number of entries in target buffer (non-negative	22
IIN		integer)	23
IN	target_datatype	datatype of each entry in target buffer (handle)	24
IN	win	window object used for communication (handle)	25
IIN	WIII	window object used for communication (nandle)	26 27
C bindi	ng		28
	_Get(void *origin_addr,	int origin count.	29
		<pre>in_datatype, int target_rank,</pre>	30
		lisp, int target_count,	31
	MPI_Datatype targ	et_datatype, MPI_Win win)	32
Fortran	2008 binding		33
		unt, origin_datatype, target_rank,	34
		et_count, target_datatype, win, ierror)	35 36
TYPI	E(*), DIMENSION(), AS	YNCHRONOUS :: origin_addr	37
		gin_count, target_rank, target_count	38
		(IN) :: origin_datatype, target_datatype	39
		IND), INTENT(IN) :: target_disp	40
	E(MPI_Win), INTENT(IN) EGER, OPTIONAL, INTENT()		41
	CGER, UPIIONAL, INIENI(42
	binding		43
MPI_GET		UNT, ORIGIN_DATATYPE, TARGET_RANK,	44
/1		ET_COUNT, TARGET_DATATYPE, WIN, IERROR)	45 46
	pe> ORIGIN_ADDR(*)	IN_DATATYPE, TARGET_RANK, TARGET_COUNT,	40
11/11	TARGET_DATATYPE,		48
		,	

 $1 \\ 2$

3

4

 $\mathbf{5}$

6 7 8

9

1 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 2 Similar to MPI_PUT, except that the direction of data transfer is reversed. Data 3 are copied from the target memory to the origin. The origin_datatype may not specify 4 overlapping entries in the origin buffer. The target buffer must be contained within the 5 target window or within attached memory in a dynamic window, and the copied data must 6 fit, without truncation, in the origin buffer. 7 8 12.3.3 Examples for Communication Calls 9 10 These examples show the use of the MPI_GET function. As all MPI RMA communication 11 functions are nonblocking, they must be completed. In the following, this is accomplished 12with the routine MPI_WIN_FENCE, introduced in Section 12.5. 13 14**Example 12.1** We show how to implement the generic indirect assignment A = B(map), 15where A, B, and map have the same distribution, and map is a permutation. To simplify, we 16assume a block distribution with equal size blocks. 17SUBROUTINE MAPVALS(A, B, map, m, comm, p) 18 19USE MPI INTEGER m, map(m), comm, p 2021REAL A(m), B(m)22 & ! used to construct origin datatypes INTEGER otype(p), oindex(m), 23& ! used to construct target datatypes 24 ttype(p), tindex(m), count(p), total(p), 25& 26disp_int, win, ierr INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint 2728! This part does the work that depends on the locations of B. 29! Can be reused while this does not change 30 31 CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr) 32 disp_int = realextent 33 34size = m * realextent CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL, 35 & comm, win, ierr) 36 37 ! This part does the work that depends on the value of map and 38 ! the locations of the arrays. 39 ! Can be reused while these do not change 4041 42! Compute number of entries to be received from each process 43 44DO i=1,p count(i) = 045END DO 46 47DO i=1,m 48 j = map(i)/m+1

Unofficial Draft for Comment Only

```
1
   count(j) = count(j)+1
                                                                                       \mathbf{2}
END DO
                                                                                       3
total(1) = 0
                                                                                       4
                                                                                       5
DO i=2,p
                                                                                       6
   total(i) = total(i-1) + count(i-1)
END DO
                                                                                       7
                                                                                       9
DO i=1,p
                                                                                       10
   count(i) = 0
                                                                                       11
END DO
                                                                                       12
! compute origin and target indices of entries.
                                                                                       13
                                                                                       14
! entry i at current process is received from location
                                                                                       15
! k at process (j-1), where map(i) = (j-1)*m + (k-1),
                                                                                       16
! j = 1...p and k = 1...m
                                                                                       17
                                                                                       18
DO i=1,m
                                                                                       19
   j = map(i)/m+1
   k = MOD(map(i), m) + 1
                                                                                       20
                                                                                      21
   count(j) = count(j)+1
   oindex(total(j) + count(j)) = i
                                                                                      22
                                                                                      23
   tindex(total(j) + count(j)) = k
                                                                                       ^{24}
END DO
                                                                                      25
                                                                                       26
! create origin and target datatypes for each get operation
DO i=1,p
                                                                                      27
   CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
                                                                                      28
                                                                                      29
                                          oindex(total(i)+1:total(i)+count(i)), &
                                                                                      30
                                         MPI_REAL, otype(i), ierr)
                                                                                      31
   CALL MPI_TYPE_COMMIT(otype(i), ierr)
                                                                                      32
   CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
                                                                                      33
                                         tindex(total(i)+1:total(i)+count(i)), &
                                                                                      34
                                         MPI_REAL, ttype(i), ierr)
   CALL MPI_TYPE_COMMIT(ttype(i), ierr)
                                                                                      35
END DO
                                                                                      36
                                                                                      37
! this part does the assignment itself
                                                                                      38
CALL MPI_WIN_FENCE(0, win, ierr)
                                                                                       39
disp_aint = 0
                                                                                       40
                                                                                      41
DO i=1,p
                                                                                      42
   CALL MPI_GET(A, 1, otype(i), i-1, disp_aint, 1, ttype(i), win, ierr)
END DO
                                                                                      43
                                                                                       44
CALL MPI_WIN_FENCE(0, win, ierr)
                                                                                       45
                                                                                       46
CALL MPI_WIN_FREE(win, ierr)
                                                                                       47
DO i=1,p
                                                                                       48
   CALL MPI_TYPE_FREE(otype(i), ierr)
```

```
1
        CALL MPI_TYPE_FREE(ttype(i), ierr)
\mathbf{2}
     END DO
3
     RETURN
4
     END
5
6
     Example 12.2 A simpler version can be written that does not require that a datatype
7
     be built for the target buffer. But, one then needs a separate get call for each entry, as
8
     illustrated below. This code is much simpler, but usually much less efficient, for large arrays.
9
10
     SUBROUTINE MAPVALS(A, B, map, m, comm, p)
11
     USE MPI
12
     INTEGER m, map(m), comm, p
13
     REAL A(m), B(m)
14
     INTEGER disp_int, win, ierr
15
     INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
16
17
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
18
     disp_int = realextent
19
     size = m * realextent
20
     CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
                                                                   &
21
                            comm, win, ierr)
22
23
     CALL MPI_WIN_FENCE(0, win, ierr)
24
     DO i=1,m
25
        j = map(i)/m
26
        disp_aint = MOD(map(i),m)
27
        CALL MPI_GET(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL, win, ierr)
28
     END DO
29
     CALL MPI_WIN_FENCE(0, win, ierr)
30
     CALL MPI_WIN_FREE(win, ierr)
^{31}
     RETURN
32
     END
33
34
     12.3.4 Accumulate Functions
35
36
37
38
39
```

It is often useful in a put operation to combine the data moved to the target process with the data that resides at that process, rather than replacing it. This will allow, for example, the accumulation of a sum by having all involved processes add their contributions to the sum variable in the memory of one process. The accumulate functions have slightly different 40semantics with respect to overlapping data accesses than the put and get functions; see 41 Section 12.7 for details. 42

- 43
- 44
- 45
- 46
- 47
- 48

	target_count, targe	rigin_count, origin_datatype, target_rank, target_disp, t_datatype, op, win)
IN	origin_addr	initial address of buffer (choice)
IN	origin_count	number of entries in buffer (non-negative integer)
IN	origin_datatype	datatype of each entry (handle)
IN	target_rank	rank of target (non-negative integer)
IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)
IN	target_count	number of entries in target buffer (non-negative integer)
IN	target_datatype	datatype of each entry in target buffer (handle)
IN	ор	reduce operation (handle)
IN	win	window object (handle)
C binding		
int MPI_A		l *origin_addr, int origin_count,
	••	igin_datatype, int target_rank,
	-	_disp, int target_count,
	MPI_Datatype ta	rget_datatype, MPI_Op op, MPI_Win win)
Fortran 2	2008 binding	
	2	prigin_count, origin_datatype, target_rank,
_		rget_count, target_datatype, op, win, ierror)
TYPE(INTENT(IN), ASYNCHRONOUS :: origin_addr
		igin_count, target_rank, target_count
		<pre>IT(IN) :: origin_datatype, target_datatype</pre>
		KIND), INTENT(IN) :: target_disp
	MPI_Op), INTENT(IN)	°
	MPI_Win), INTENT(IN)	-
	ER, OPTIONAL, INTENT	
D		
Fortran b	-	
MPI_ACCUM		RIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
/ +	*	RGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
• 1	> ORIGIN_ADDR(*)	
INTEG		GIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
ተእነጥር ሳ		C, OP, WIN, IERROR
INTEG	ER(KIND=MPI_ADDRESS_	KIND) TAKGET_DISP
Accun	nulate the contents of the	e origin buffer (as defined by origin_addr, origin_count, and
		ified by arguments target_count and target_datatype, at

offset target_disp, in the target window specified by target_rank and win, using the operation

```
1
     op. This is like MPI_PUT except that data is combined into the target area instead of
\mathbf{2}
     overwriting it.
3
          Any of the predefined operations for MPI_REDUCE can be used. User-defined functions
4
     cannot be used. For example, if op is MPI_SUM, each element of the origin buffer is added
\mathbf{5}
     to the corresponding element in the target, replacing the former value in the target.
6
          Each datatype argument must be a predefined datatype or a derived datatype, where
7
     all basic components are of the same predefined datatype. Both datatype arguments must
8
     be constructed from the same predefined datatype. The operation op applies to elements of
9
     that predefined type. The parameter target_datatype must not specify overlapping entries,
10
     and the target buffer must fit in the target window.
11
          A new predefined operation, MPI_REPLACE, is defined. It corresponds to the associative
12
     function f(a,b) = b; i.e., the current value in the target memory is replaced by the value
13
     supplied by the origin.
14
          MPI_REPLACE can be used only in MPI_ACCUMULATE, MPI_RACCUMULATE,
15
     MPI_GET_ACCUMULATE, MPI_FETCH_AND_OP, and MPI_RGET_ACCUMULATE, but not
16
     in collective reduction operations such as MPI_REDUCE.
17
           Advice to users.
                             MPI_PUT is a special case of MPI_ACCUMULATE, with the op-
18
           eration MPI_REPLACE. Note, however, that MPI_PUT and MPI_ACCUMULATE have
19
           different constraints on concurrent updates. (End of advice to users.)
20
21
22
     Example 12.3 We want to compute B(j) = \sum_{map(i)=j} A(i). The arrays A, B, and map
23
     are distributed in the same manner. We write the simple version.
24
25
     SUBROUTINE SUM(A, B, map, m, comm, p)
26
     USE MPI
27
     INTEGER m, map(m), comm, p, win, ierr, disp_int
28
     REAL A(m), B(m)
29
     INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
30
^{31}
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
32
     size = m * realextent
33
     disp_int = realextent
34
     CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL, &
35
                            comm, win, ierr)
36
37
     CALL MPI_WIN_FENCE(0, win, ierr)
38
     DO i=1,m
39
        j = map(i)/m
40
        disp_aint = MOD(map(i),m)
41
        CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL,
                                                                                      &
42
                               MPI_SUM, win, ierr)
43
     END DO
^{44}
     CALL MPI_WIN_FENCE(0, win, ierr)
45
46
     CALL MPI_WIN_FREE(win, ierr)
47
     RETURN
48
     END
```

This code is identical to the code in Example 12.2, except that a call to get has been replaced by a call to accumulate. (Note that, if map is one-to-one, the code computes $B = A(map^{-1})$, which is the reverse assignment to the one computed in that previous example.) In a similar manner, we can replace in Example 12.1, the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

Get Accumulate Function

It is often useful to have fetch-and-accumulate semantics such that the remote data is returned to the caller before the sent data is accumulated into the remote data. The get and accumulate steps are executed atomically for each basic element in the datatype (see Section 12.7 for details). The predefined operation MPI_REPLACE provides fetch-and-set behavior.

MPI_GET_ACCUMULATE(origin_addr, origin_count, origin_datatype, result_addr,				
result_count, result_datatype, target_rank, target_disp, target_count,				
	target_datatype, op, win)			
IN	origin_addr	initial address of buffer (choice)	19 20	
	-		20 21	
IN	origin_count	number of entries in origin buffer (non-negative	21	
		integer)	23	
IN	origin_datatype	datatype of each entry in origin buffer (handle)	24	
OUT	result_addr	initial address of result buffer (choice)	25	
IN	result_count	number of entries in result buffer (non-negative	26	
		integer)	27	
IN	result_datatype	datatype of each entry in result buffer (handle)	28	
			29	
IN	target_rank	rank of target (non-negative integer)	30	
IN	target_disp	displacement from start of window to beginning of	31	
		target buffer (non-negative integer)	32	
IN	target_count	number of entries in target buffer (non-negative	33	
		integer)	34	
		<i>o</i> ,	35	
IN	target_datatype	datatype of each entry in target buffer (handle)	36	
IN	ор	reduce operation (handle)	37	
IN	win	window object (handle)	38	
			39 40	
C binding				
int MPI_Get_accumulate(const void *origin_addr, int origin_count,				
Int mildet accumulate (const volu *origin_auur, int origin_count,				

(

i MPI_Datatype origin_datatype, void *result_addr, 43 int result_count, MPI_Datatype result_datatype, 44int target_rank, MPI_Aint target_disp, int target_count, 45MPI_Datatype target_datatype, MPI_Op op, MPI_Win win) 46

Fortran 2008 binding

Unofficial Draft for Comment Only

1 $\mathbf{2}$

3

4

56

7 8

9

10

11

12

13

141516

47

1	MDT Cot accumulate (entrin addr. entrin count entrin deteture recult addr.
2	MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
	result_count, result_datatype, target_rank, target_disp,
3	<pre>target_count, target_datatype, op, win, ierror)</pre>
4	<pre>TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr</pre>
5	<pre>INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,</pre>
6	target_count
7	TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
8	target_datatype
9	TYPE(*), DIMENSION(), ASYNCHRONOUS :: result_addr
10	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
11	TYPE(MPI_Op), INTENT(IN) :: op
12	TYPE(MPI_Win), INTENT(IN) :: win
13	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14	
15	Fortran binding
16	MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
17	RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
18	TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
19	<type> ORIGIN_ADDR(*), RESULT_ADDR(*)</type>
20	INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
21	TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
22	INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
23	
24	Accumulate origin_count elements of type origin_datatype from the origin buffer (
25	origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank
	and win, using the operation op and return in the result buffer result_addr the content
26	of the target buffer before the accumulation, specified by target_disp, target_count, and
27	target_datatype. The data transferred from origin to target must fit, without truncation,
28	in the target buffer. Likewise, the data copied from target to origin must fit, without
29	truncation, in the result buffer.
30	The origin and result buffers (origin_addr and result_addr) must be disjoint. Each
31	datatype argument must be a predefined datatype or a derived datatype where all basic
32	components are of the same predefined datatype. All datatype arguments must be con-
33	structed from the same predefined datatype. The operation op applies to elements of that
34	predefined type. target_datatype must not specify overlapping entries, and the target buffer
35	must fit in the target window or in attached memory in a dynamic window. The operation
36	is executed atomically for each basic datatype; see Section 12.7 for details.
37	Any of the predefined operations for MPI_REDUCE, as well as MPI_NO_OP or
38	MPI_REPLACE can be specified as op. User-defined functions cannot be used. A new
39	predefined operation, MPI_NO_OP, is defined. It corresponds to the associative function
40	f(a,b) = a; i.e., the current value in the target memory is returned in the result buffer at
41	the origin and no operation is performed on the target buffer. When MPI_NO_OP is specified
42	as the operation, the origin_addr, origin_count, and origin_datatype arguments are ignored.
43	MPI_NO_OP can be used only in MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE,
44	and MPI_FETCH_AND_OP. MPI_NO_OP cannot be used in MPI_ACCUMULATE,
45	MPI_RACCUMULATE, or collective reduction operations, such as MPI_REDUCE and others.

46

47 Advice to users. MPI_GET is similar to MPI_GET_ACCUMULATE, with the opera-48 tion MPI_NO_OP. Note, however, that MPI_GET and MPI_GET_ACCUMULATE have

Unofficial Draft for Comment Only

different constraints on concurrent updates. (End of advice to users.)

Fetch and Op Function

The generic functionality of MPI_GET_ACCUMULATE might limit the performance of fetchand-increment or fetch-and-add calls that might be supported by special hardware operations. MPI_FETCH_AND_OP thus allows for a fast implementation of a commonly used subset of the functionality of MPI_GET_ACCUMULATE.

MPI_FETCH_AND_OP(origin_addr, result_addr, datatype, target_rank, target_disp, op, win)				
			11	
			12	
IN	origin_addr	initial address of buffer (choice)	13	
OUT	result_addr	initial address of result buffer (choice)	14	
IN	datatype	datatype of the entry in origin, result, and target	15	
	51	buffers (handle)	16	
	La constance a l		17	
IN	target_rank	rank of target (non-negative integer)	18	
IN	target_disp	displacement from start of window to beginning of	19	
		target buffer (non-negative integer)	20	
IN	ор	reduce operation (handle)	21	
	op		22	
IN	win	window object (handle)	23	
			24	
C binding	<u>,</u>		25	

C binding

<pre>int MPI_Fetch_and_op(const void *origin_addr, void *result_addr,</pre>	26		
MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,	27		
MPI_Op op, MPI_Win win)	28		
Fortran 2008 binding	29		
J	30		
MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,			
target_disp, op, win, ierror)	32		
<pre>TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr</pre>	33		
TYPE(*), DIMENSION(), ASYNCHRONOUS :: result_addr	34		
TYPE(MPI_Datatype), INTENT(IN) :: datatype	35		
INTEGER, INTENT(IN) :: target_rank	36		
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp			
TYPE(MPI_Op), INTENT(IN) :: op	37		
TYPE(MPI_Win), INTENT(IN) :: win	38 39		
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
	40		
Fortran binding	41		
MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,			
TARGET_DISP, OP, WIN, IERROR)	43		
<type> ORIGIN_ADDR(*), RESULT_ADDR(*)</type>	44		
INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR	45		
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	46		
	47		
	48		

 $\mathbf{2}$

 $\mathbf{5}$

 $\overline{7}$

Accumulate one element of type datatype from the origin buffer (origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank and win, using the operation op and return in the result buffer result_addr the content of the target buffer before the accumulation.

⁵ The origin and result buffers (origin_addr and result_addr) must be disjoint. Any of the ⁶ predefined operations for MPI_REDUCE, as well as MPI_NO_OP or MPI_REPLACE, can be ⁷ specified as op; user-defined functions cannot be used. The datatype argument must be a ⁸ predefined datatype. The operation is executed atomically.

¹⁰ Compare and Swap Function

Another useful operation is an atomic compare and swap where the value at the origin is compared to the value at the target, which is atomically replaced by a third value only if the values at origin and target are equal.

```
16
     MPI_COMPARE_AND_SWAP(origin_addr, compare_addr, result_addr, datatype,
17
                    target_rank, target_disp, win)
18
                                             initial address of buffer (choice)
       IN
                 origin_addr
19
20
       IN
                 compare_addr
                                             initial address of compare buffer (choice)
21
       OUT
                 result_addr
                                             initial address of result buffer (choice)
22
                                             datatype of the element in all buffers (handle)
       IN
                 datatype
23
^{24}
       IN
                 target_rank
                                              rank of target (non-negative integer)
25
       IN
                 target_disp
                                              displacement from start of window to beginning of
26
                                              target buffer (non-negative integer)
27
       IN
                                              window object (handle)
                 win
28
29
     C binding
30
     int MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr,
^{31}
                     void *result_addr, MPI_Datatype datatype, int target_rank,
32
                    MPI_Aint target_disp, MPI_Win win)
33
34
     Fortran 2008 binding
35
     MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype,
36
                    target_rank, target_disp, win, ierror)
37
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr,
38
                     compare_addr
39
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
40
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
          INTEGER, INTENT(IN) :: target_rank
42
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
43
          TYPE(MPI_Win), INTENT(IN) :: win
44
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
```

```
46 Fortran binding
```

```
    MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
    TARGET_RANK, TARGET_DISP, WIN, IERROR)
```

9

11

<type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*) INTEGER DATATYPE, TARGET_RANK, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP

This function compares one element of type datatype in the compare buffer compare_addr with the buffer at offset target_disp in the target window specified by target_rank and win and replaces the value at the target with the value in the origin buffer origin_addr if the compare buffer and the target buffer are identical. The original value at the target is returned in the buffer result_addr. The parameter datatype must belong to one of the following categories of predefined datatypes: C integer, Fortran integer, Logical, Multi-language types, or Byte as specified in Section 6.9.2. The origin and result buffers (origin_addr and result_addr) must be disjoint.

12.3.5 Request-based RMA Communication Operations

Request-based RMA communication operations allow the user to associate a request handle with the RMA operations and test or wait for the completion of these requests using the functions described in Section 3.7.3. Request-based RMA operations are only valid within a passive target epoch (see Section 12.5).

Upon returning from a completion call in which an RMA operation completes, all fields of the status object, if any, and the results of status query functions (e.g., MPI_GET_COUNT) are undefined with the exception of MPI_ERROR if appropriate (see Section 3.2.5). It is valid to mix different request types (e.g., any combination of RMA requests, collective requests, I/O requests, generalized requests, or point-to-point requests) in functions that enable multiple completions (e.g., MPI_WAITALL). It is erroneous to call

MPI_REQUEST_FREE or MPI_CANCEL for a request associated with an RMA operation. RMA requests are not persistent.

The end of the epoch, or explicit bulk synchronization using MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL, or MPI_WIN_FLUSH_LOCAL_ALL, also indicates completion of the RMA operations. However, users must still wait or test on the request handle to allow the MPI implementation to clean up any resources associated with these requests; in such cases the wait operation will complete locally. $\mathbf{2}$

 24

12	Wi I_IN OT (Orgin_addi, Orgin_count, Orgin_datatype, target_rank, target_disp,				
$\frac{3}{4}$	IN	origin_addr	initial address of origin buffer (choice)		
5	IN	origin_count	number of entries in origin buffer (non-negative integer)		
7	IN	origin_datatype	datatype of each entry in origin buffer (handle)		
8 9	IN	target_rank	rank of target (non-negative integer)		
9 10 11	IN	target_disp	displacement from start of window to target buffer (non-negative integer)		
12 13	IN	target_count	number of entries in target buffer (non-negative integer)		
14 15	IN	target_datatype	datatype of each entry in target buffer (handle)		
16	IN	win	window object used for communication (handle)		
17	OUT	request	RMA request (handle)		
18 19					
20	C bindin	0	adda int anigin sount		
21	<pre>int MPI_Rput(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank,</pre>				
22		v. 0	isp, int target_count,		
23 24		MPI_Datatype targ	et_datatype, MPI_Win win,		
24 25		MPI_Request *requ	est)		
26	Fortran	2008 binding			
27	MPI_Rput		ount, origin_datatype, target_rank,		
28					
29 30	тург	ierror) F(*) DIMENSION() INT	TENT(IN), ASYNCHRONOUS :: origin_addr		
30 31			gin_count, target_rank, target_count		
32					
33	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp				
34					
35	TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
$\frac{36}{37}$					
38	Fortran binding				
39	MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,				
40		IERROR)	EI_COUNI, IARGEI_DAIAIIFE, WIN, REQUESI,		
41	<typ< th=""><th><pre>De> ORIGIN_ADDR(*)</pre></th><th></th></typ<>	<pre>De> ORIGIN_ADDR(*)</pre>			
42	INTE	CGER ORIGIN_COUNT, ORIG	IN_DATATYPE, TARGET_RANK, TARGET_COUNT,		
43 44			WIN, REQUEST, IERROR		
45	INTE	EGER(KIND=MPI_ADDRESS_K)	IND) TARGET_DISP		
46			JT (Section 12.3.1), except that it allocates a commu-		
47	nication 1	request object and associat	es it with the request handle (the argument request).		
48					

The completion of an MPI_RPUT operation (i.e., after the corresponding test or wait) indicates that the sender is now free to update the locations in the origin buffer. It does not indicate that the data is available at the target window. If remote completion is required, MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_UNLOCK, or MPI_WIN_UNLOCK_ALL can be used.

MPI_RGET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, request)

OUT	origin_addr	initial address of origin buffer (choice)	10
IN	origin_count	number of entries in origin buffer (non-negative integer)	11 12
IN	origin_datatype	datatype of each entry in origin buffer (handle)	$13 \\ 14$
IN	target_rank	rank of target (non-negative integer)	15
IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)	16 17
IN	target_count	number of entries in target buffer (non-negative integer)	18 19 20
IN	target_datatype	datatype of each entry in target buffer (handle)	21
IN	win	window object used for communication (handle)	22 23
OUT	request	RMA request (handle)	23 24

C binding

int MPI_Rget(void *origin_addr, int origin_count,	27
MPI_Datatype origin_datatype, int target_rank,	28
MPI_Aint target_disp, int target_count,	29
MPI_Datatype target_datatype, MPI_Win win,	30
MPI_Request *request)	31
Fortran 2008 binding	32
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,	33
	34
<pre>target_disp, target_count, target_datatype, win, request, ierror)</pre>	35
	36
TYPE(*), DIMENSION(), ASYNCHRONOUS :: origin_addr	37
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count	38
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype	39
<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp</pre>	40
TYPE(MPI_Win), INTENT(IN) :: win	41
TYPE(MPI_Request), INTENT(OUT) :: request	42
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	43
Fortran binding	44
MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	45
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,	46
IERROR)	47
<type> ORIGIN_ADDR(*)</type>	48
() PEC OUT OT ADDI((*)	

	538	CH	IAPTER 12.	ONE-SIDED COMMUNICATIONS	
1 2 3		ER ORIGIN_COUNT, ORIGIN_D TARGET_DATATYPE, WIN ER(KIND=MPI_ADDRESS_KIND)	I, REQUEST,	IERROR	
4 5 7 8 9 10	nication re- that can be indicates the	MPI_RGET is similar to MPI_GET (Section 12.3.2), except that it allocates a commu- cation request object and associates it with the request handle (the argument request) hat can be used to wait or test for completion. The completion of an MPI_RGET operation dicates that the data is available in the origin buffer. If origin_addr points to memory tached to a window, then the data becomes available in the private copy of this window.			
11 12	MPI_RACC	UMULATE(origin_addr, origin_ target_count, target_data	-	_datatype, target_rank, target_disp, request)	
$13 \\ 14$	IN	origin_addr	initial address	s of buffer (choice)	
15	IN	origin_count	number of ent	tries in buffer (non-negative integer)	
16	IN	origin_datatype	datatype of e	ach entry in origin buffer (handle)	
17 18	IN	target_rank	rank of target	(non-negative integer)	
19 20	IN	target_disp	-	from start of window to beginning of (non-negative integer)	
21 22 23	IN	target_count	number of ent integer)	tries in target buffer (non-negative	
23 24	IN	target_datatype	datatype of ea	ach entry in target buffer (handle)	
25	IN	ор	reduce operat	ion (handle)	
26	IN	win	window objec	t (handle)	
27 28 29	OUT	request	RMA request	(handle)	
29 30	C binding				
31	int MPI_Ra	accumulate(const void *or	-	-	
32		MPI_Datatype origin_c		-	
33 34		MPI_Aint target_disp	•	/_count, PI_Op op, MPI_Win win,	
35		MPI_Request *request		• • • ·,	
36	Fortran 2	008 binding			
37			n_count, or	igin_datatype, target_rank,	
38				et_datatype, op, win, request,	
39 40	/	ierror)			
41	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count				
42		0	•	datatype, target_datatype	
43		ER(KIND=MPI_ADDRESS_KIND)	-		
44 45		MPI_Op), INTENT(IN) :: op			
46		MPI_Win), INTENT(IN) :: w			
47		MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT)	-		
48	101100	, o. 1100002, 1011001 (001)			

Unofficial Draft for Comment Only

MPI_RA <t IN IN MF it alloca argume MPI_RA</t 	TARGET_DISP, TA IERROR) ype> ORIGIN_ADDR(*) TEGER ORIGIN_COUNT, OR TARGET_DATATYP TEGER(KIND=MPI_ADDRESS PI_RACCUMULATE is simulates a communication requises ACCUMULATE operation is	ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, ARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, IGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, E, OP, WIN, REQUEST, IERROR E_KIND) TARGET_DISP ilar to MPI_ACCUMULATE (Section 12.3.4), except that test object and associates it with the request handle (the sed to wait or test for completion. The completion of an indicates that the origin buffer is free to be updated. It ion has completed at the target window.	1 2 3 4 5 6 7 8 9 10 11 12 13 14
MPI R	GET ACCUMULATE(origin	_addr, origin_count, origin_datatype, result_addr,	15 16
	result_count, resul	t_datatype, target_rank, target_disp, target_count,	17 18
	target_datatype, c		19
IN	origin_addr	initial address of buffer (choice)	20
IN	origin_count	number of entries in origin buffer (non-negative integer)	21 22
IN	origin_datatype	datatype of each entry in origin buffer (handle)	23
OUT	result_addr	initial address of result buffer (choice)	24 25
IN	result_count	number of entries in result buffer (non-negative	26
		integer)	27
IN	result_datatype	datatype of entries in result buffer (handle)	28
IN	target_rank	rank of target (non-negative integer)	29 30
IN	target_disp	displacement from start of window to beginning of	31 32
		target buffer (non-negative integer)	33
IN	target_count	number of entries in target buffer (non-negative integer)	34
IN	target_datatype	datatype of each entry in target buffer (handle)	35
IN			36
	op	reduce operation (handle)	37 38
IN	win	window object (handle)	39
OUT	request	RMA request (handle)	40
C bind	ling		41
		t void *origin_addr, int origin_count,	42 43
	•	rigin_datatype, void *result_addr,	44
		nt, MPI_Datatype result_datatype,	45
	-	x, MPI_Aint target_disp, int target_count,	46
	MPI_Datatype ta	arget_datatype, MPI_Op op, MPI_Win win,	47

MPI_Request *request)

Unofficial Draft for Comment Only

1 Fortran 2008 binding $\mathbf{2}$ MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype, 3 result_addr, result_count, result_datatype, target_rank, 4 target_disp, target_count, target_datatype, op, win, request, 5ierror) 6 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 7 INTEGER, INTENT(IN) :: origin_count, result_count, target_rank, 8 target_count 9 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype, 10 target_datatype 11 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr 12INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 13 TYPE(MPI_Op), INTENT(IN) :: op 14TYPE(MPI_Win), INTENT(IN) :: win 15TYPE(MPI_Request), INTENT(OUT) :: request 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17 Fortran binding 18 MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, 19 RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, 20TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, 21IERROR) 22 <type> ORIGIN_ADDR(*), RESULT_ADDR(*) 23INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE, 24TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, 25IERROR 26INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 2728MPI_RGET_ACCUMULATE is similar to MPI_GET_ACCUMULATE (Section 12.3.4),

²³ MPI_RGET_ACCOMOLATE is similar to MPI_GET_ACCOMOLATE (Section 12.3.4), ²⁹ except that it allocates a communication request object and associates it with the request ³⁰ handle (the argument request) that can be used to wait or test for completion. The com-³¹ pletion of an MPI_RGET_ACCUMULATE operation indicates that the data is available in ³² the result buffer and the origin buffer is free to be updated. It does not indicate that the ³³ operation has been completed at the target window.

12.4 Memory Model

35

36

37 The memory semantics of RMA are best understood by using the concept of *public* and 38 private window copies. We assume that systems have a public memory region that is 39 addressable by all processes (e.g., the shared memory in shared memory machines or the 40 exposed main memory in distributed memory machines). In addition, most machines have 41 fast private buffers (e.g., transparent caches or explicit communication buffers) local to 42each process where copies of data elements from the main memory can be stored for faster 43 access. Such buffers are either coherent, i.e., all updates to main memory are reflected in 44 all private copies consistently, or non-coherent, i.e., conflicting accesses to main memory 45 need to be synchronized and updated in all private copies explicitly. Coherent systems 46 allow direct updates to remote memory without any participation of the remote side. Non-47 coherent systems, however, need to call RMA functions in order to reflect updates to the 48

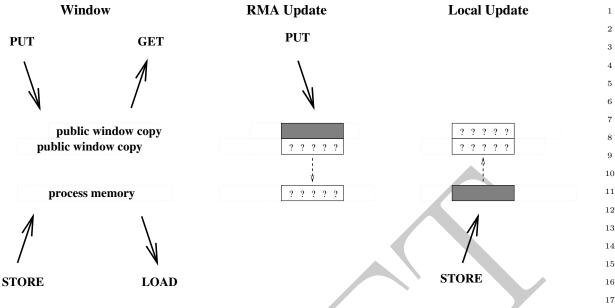


Figure 12.1: Schematic description of the public/private window operations in the MPI_WIN_SEPARATE memory model for two overlapping windows.

public window in their private memory. Thus, in coherent memory, the public and the private window are identical while they remain logically separate in the non-coherent case. MPI thus differentiates between two **memory models** called **RMA unified**, if public and private window are logically identical, and **RMA separate**, otherwise.

In the RMA separate model, there is only one instance of each variable in process memory, but a distinct *public* copy of the variable for each window that contains it. A load accesses the instance in process memory (this includes MPI sends). A local store accesses and updates the instance in process memory (this includes MPI receives), but the update may affect other public copies of the same locations. A get on a window accesses the public copy of that window. A put or accumulate on a window accesses and updates the public copy of that window, but the update may affect the private copy of the same locations in process memory, and public copies of other overlapping windows. This is illustrated in Figure 12.1.

In the RMA unified model, public and private copies are identical and updates via put or accumulate calls are eventually observed by load operations without additional RMA calls. A store access to a window is eventually visible to remote get or accumulate calls without additional RMA calls. These stronger semantics of the RMA unified model allow the user to omit some synchronization calls and potentially improve performance.

Advice to users. If accesses in the RMA unified model are not synchronized (with locks or flushes, see Section 12.5.3), load and store operations might observe changes to the memory while they are in progress. The order in which data is written is not specified unless further synchronization is used. This might lead to inconsistent views on memory and programs that assume that a transfer is complete by only checking parts of the message are erroneous. (*End of advice to users.*)

The memory model for a particular RMA window can be determined by accessing the attribute MPI_WIN_MODEL. If the memory model is the unified model, the value of this attribute is MPI_WIN_UNIFIED; otherwise, the value is MPI_WIN_SEPARATE.

Unofficial Draft for Comment Only

18

19 20

21

22

23

 24

25

26

27

28

29

30

 31

32

33

34

35

36

37

38 39

40

41

42

43

44

 $45 \\ 46$

47

1 2

> 3 4

5

6

7

8

9 10

11

12

13

14

15

16

17

12.5 Synchronization Calls

RMA communications fall in two categories:

• active target communication, where data is moved from the memory of one process to the memory of another, and both are explicitly involved in the communication. This communication pattern is similar to message passing, except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization.

• **passive target communication**, where data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer. Thus, two origin processes may communicate by accessing the same location in a target window. The process that owns the target window may be distinct from the two communicating processes, in which case it does not participate explicitly in the communication. This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, irrespective of location.

RMA communication calls with argument win must occur at a process only within an **access epoch** for win. Such an epoch starts with an RMA synchronization call on win; it proceeds with zero or more RMA communication calls (e.g., MPI_PUT, MPI_GET or MPI_ACCUMULATE) on win; it completes with another synchronization call on win. This allows users to amortize one synchronization with multiple data transfers and provide implementors more flexibility in the implementation of RMA operations.

Distinct access epochs for win at the same process must be disjoint. On the other hand, epochs pertaining to different win arguments may overlap. Local operations or other MPI calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations 27only within an **exposure epoch**. Such an epoch is started and completed by RMA syn-28chronization calls executed by the target process. Distinct exposure epochs at a process on 29 the same window must be disjoint, but such an exposure epoch may overlap with exposure 30 epochs on other windows or with access epochs for the same or other win arguments. There 31 is a one-to-one matching between access epochs at origin processes and exposure epochs 32 on target processes: RMA operations issued by an origin process for a target window will 33 access that target window during the same exposure epoch if and only if they were issued 34 during the same access epoch. 35

In passive target communication the target process does not execute RMA synchronization calls, and there is no concept of an exposure epoch.

38 39

40

41

42

43

44

45

MPI provides three synchronization mechanisms:

1. The MPI_WIN_FENCE collective synchronization call supports a simple synchronization pattern that is often used in parallel computations: namely a loosely-synchronous model, where global computation phases alternate with global communication phases. This mechanism is most useful for loosely synchronous algorithms where the graph of communicating processes changes very frequently, or where each process communicates with many others.

This call is used for active target communication. An access epoch at an origin process or an exposure epoch at a target process are started and completed by calls to MPI_WIN_FENCE. A process can access windows at all processes in the group of win

Unofficial Draft for Comment Only

during such an access epoch, and the local window can be accessed by all processes in the group of win during such an exposure epoch.

2. The four functions MPI_WIN_START, MPI_WIN_COMPLETE, MPI_WIN_POST, and MPI_WIN_WAIT can be used to restrict synchronization to the minimum: only pairs of communicating processes synchronize, and they do so only when a synchronization is needed to order correctly RMA accesses to a window with respect to local accesses to that same window. This mechanism may be more efficient when each process communicates with few (logical) neighbors, and the communication graph is fixed or changes infrequently.

These calls are used for active target communication. An access epoch is started at the origin process by a call to MPI_WIN_START and is terminated by a call to MPI_WIN_COMPLETE. The start call has a group argument that specifies the group of target processes for that epoch. An exposure epoch is started at the target process by a call to MPI_WIN_POST and is completed by a call to MPI_WIN_WAIT. The post call has a group argument that specifies the set of origin processes for that epoch.

3. Finally, shared lock access is provided by the functions MPI_WIN_LOCK, MPI_WIN_LOCK_ALL, MPI_WIN_UNLOCK, and MPI_WIN_UNLOCK_ALL. MPI_WIN_LOCK and MPI_WIN_UNLOCK also provide exclusive lock capability. Lock synchronization is useful for MPI applications that emulate a shared memory model via MPI calls; e.g., in a "billboard" model, where processes can, at random times, access or update different parts of the billboard.

These four calls provide passive target communication. An access epoch is started by a call to MPI_WIN_LOCK or MPI_WIN_LOCK_ALL and terminated by a call to MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL, respectively.

Figure 12.2 illustrates the general synchronization pattern for active target communication. The synchronization between **post** and **start** ensures that the put call of the origin process does not start until the target process exposes the window (with the **post** call); the target process will expose the window only after preceding local accesses to the window have completed. The synchronization between **complete** and **wait** ensures that the put call of the origin process completes before the window is unexposed (with the **wait** call). The target process will execute following local accesses to the target window only after the **wait** returned.

Figure 12.2 shows operations occurring in the natural temporal order implied by the synchronizations: the post occurs before the matching start, and complete occurs before the matching wait. However, such strong synchronization is more than needed for correct ordering of window accesses. The semantics of MPI calls allow weak synchronization, as illustrated in Figure 12.3. The access to the target window is delayed until the window is exposed, after the post. However the start may complete earlier; the put and complete may also terminate earlier, if put data is buffered by the implementation. The synchronization calls order correctly window accesses, but do not necessarily synchronize other operations. This weaker synchronization semantic allows for more efficient implementations.

Figure 12.4 illustrates the general synchronization pattern for passive target communication. The first origin process communicates data to the second origin process, through the memory of the target process; the target process is not explicitly involved in the communication. The lock and unlock calls ensure that the two RMA accesses do not occur 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11

12

13

14

15

16 17

18

19

20

21

22

23

 24

25

26

27 28

29

30

 31

32

33

34

35

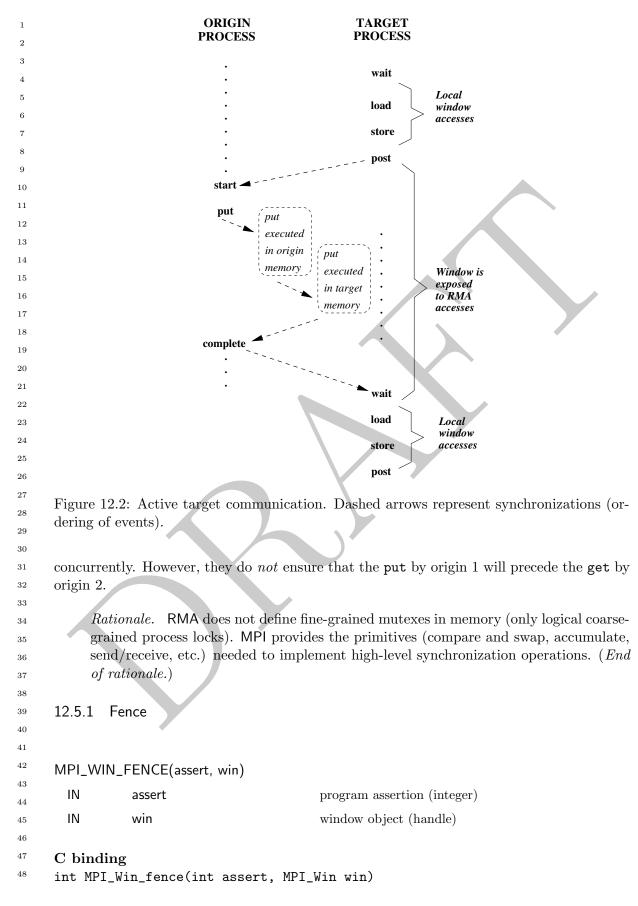
36 37

38

39

40 41

42



Unofficial Draft for Comment Only

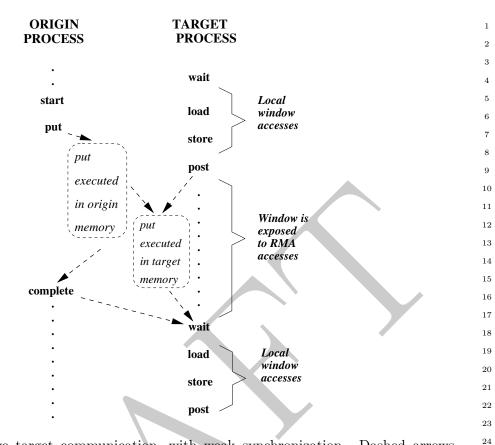


Figure 12.3: Active target communication, with weak synchronization. Dashed arrows represent synchronizations (ordering of events)

Fortran 2008 binding MPI_Win_fence(assert, win, ierror) INTEGER, INTENT(IN) :: assert TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_WIN_FENCE(ASSERT, WIN, IERROR) INTEGER ASSERT, WIN, IERROR

The MPI call MPI_WIN_FENCE(assert, win) synchronizes RMA calls on win. The call is collective on the group of win. All RMA operations on win originating at a given process and started before the fence call will complete at that process before the fence call returns. They will be completed at their target before the fence call returns at the target. RMA operations on win started by a process after the fence call returns will access their target window only after MPI_WIN_FENCE has been called by the target process.

The call completes an RMA access epoch if it was preceded by another fence call and the local process issued RMA communication calls on win between these two calls. The call completes an RMA exposure epoch if it was preceded by another fence call and the local window was the target of RMA accesses between these two calls. The call starts an RMA access epoch if it is followed by another fence call and by RMA communication calls issued between these two fence calls. The call starts an exposure epoch if it is followed by another 48

Unofficial Draft for Comment Only

25

26 27

28

29

30

31

32 33

34

35

36

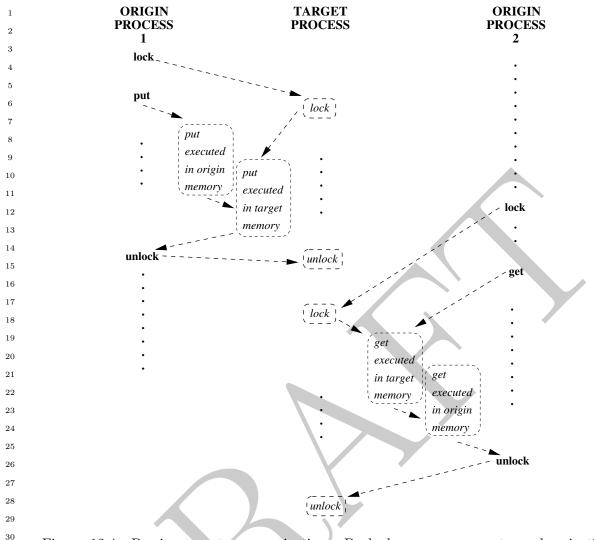
37

38

39

40

41



³⁰ Figure 12.4: Passive target communication. Dashed arrows represent synchronizations ³¹ (ordering of events).

fence call and the local window is the target of RMA accesses between these two fence calls. Thus, the fence call is equivalent to calls to a subset of post, start, complete, wait.

A fence call usually entails a barrier synchronization: a process completes a call to MPI_WIN_FENCE only after all other processes in the group entered their matching call. However, a call to MPI_WIN_FENCE that is known not to end any epoch (in particular, a call with assert equal to MPI_MODE_NOPRECEDE) does not necessarily act as a barrier.

The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 12.5.5. A value of assert = 0 is always valid.

Advice to users. Calls to MPI_WIN_FENCE should both precede and follow calls to RMA communication functions that are synchronized with fence calls. (*End of advice to users.*)

 $\frac{46}{47}$

43

44

45

33

12.5.2 General Active Target Synchronization

MPI_WIN_START(group, assert, win)

IN	group	group of target processes (handle)
IN	assert	program assertion (integer)
IN	win	window object (handle)

C binding

int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)

Fortran 2008 binding

```
MPI_Win_start(group, assert, win, ierror)
   TYPE(MPI_Group), INTENT(IN) :: group
   INTEGER, INTENT(IN) :: assert
   TYPE(MPI_Win), INTENT(IN) :: win
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

MPI_WIN_START(GROUP, ASSERT, WIN, IERROR) INTEGER GROUP, ASSERT, WIN, IERROR

Starts an RMA access epoch for win, RMA calls issued on win during this epoch must access only windows at processes in group. Each process in group must issue a matching call to MPI_WIN_POST. RMA accesses to each target window will be delayed, if necessary, until the target process executed the matching call to MPI_WIN_POST. MPI_WIN_START is allowed to block until the corresponding MPI_WIN_POST calls are executed, but is not required to.

The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 12.5.5. A value of assert =0 is always valid.

```
MPI_WIN_COMPLETE(win)
         win
```

window object (handle)

C binding

ĬΝ

int MPI_Win_complete(MPI_Win win)

```
Fortran 2008 binding
```

MPI_Win_complete(win, ierror) TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

```
MPI_WIN_COMPLETE(WIN, IERROR)
    INTEGER WIN, IERROR
```

5 6

9 10

11

12

13

14

15

1617

18

19

20

21

22 23

 24

25

26

2728

2930

31

32 33

34 35

36 37

38

39

40

41

42

43 44

45

46

1 2	-	•	-	on win started by a call to MPI_WIN_START. All during this epoch will have completed at the origin
3	when the	call returns.		
4	MPI_	WIN_COMPLE	TE enforces con	mpletion of preceding RMA calls at the origin, but
5	not at the	e target. A put	or accumulate	call may not have completed at the target when it
6	has compl	leted at the ori	gin.	
7	Const	ider the sequer	ice of calls in th	e example below.
8 9	Example	e 12.4 Use of	MPI_WIN_STAF	RT and MPI_WIN_COMPLETE.
10	MPT Win	start(group,	flag win).	
11		, win);	1106, 111/,	
12		complete(win).	
13	···· + _ •• + + +	comprese(win	,	
14	The o	call to MPI_W	IN_COMPLETE	does not return until the put call has completed
15	at the ori	gin; and the ta	arget window w	ill be accessed by the put operation only after the
16	call to M	PI_WIN_STAR	T has matched	a call to MPI_WIN_POST by the target process.
17	This still	leaves much c	hoice to implen	nentors. The call to MPI_WIN_START can block
18	until the	matching call	to MPI_WIN_P	OST occurs at all target processes. One can also
19	have impl	ementations w	here the call to	MPI_WIN_START is nonblocking, but the call to
20	MPI_PUT	blocks until t	he matching ca	ll to MPI_WIN_POST occurs; or implementations
21	where the	e first two call	s are nonblocki	ng, but the call to MPI_WIN_COMPLETE blocks
22				red; or even implementations where all three calls
23	can comp	lete before any	target process	has called MPI_WIN_POST — the data put must
24		,	,	llow the put to complete at the origin ahead of its
25	-	0	,	the call to MPI_WIN_POST is issued, the sequence
26	above mu	st complete, w	ithout further d	ependencies.
27				
28 29	MPI_WIN	_POST(group,	assert, win)	
30 31	IN	group		group of origin processes (handle)
32	IN	assert		program assertion (integer)
33	IN	win		window object (handle)
34				
35	C bindin	ıg		
36	int MPI_	Win_post(MPI	_Group group,	int assert, MPI_Win win)
37	Deter			
38		2008 binding		
39			assert, win,	
40		-	INTENT(IN) :	: group
41		•	IN) :: assert	
42			NTENT(IN) ::	
43		GER, UPIIUNA	L, INTENT(OUT) :: ierror
44	Fortran	binding		
45	MPI_WIN_	POST(GROUP,	ASSERT, WIN,	IERROR)
46	INTE	GER GROUP, A	SSERT, WIN, I	ERROR
47				
48				

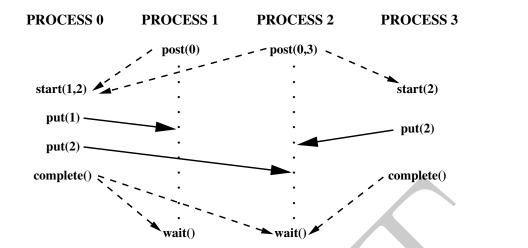


Figure 12.5: Active target communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

Starts an RMA exposure epoch for the local window associated with win. Only processes in group should access the window with RMA calls on win during this epoch. Each process in group must issue a matching call to MPI_WIN_START. MPI_WIN_POST does not block.

```
MPI_WIN_WAIT(win)
    IN win window object (handle)
C binding
int MPI_Win_wait(MPI_Win win)
Fortran 2008 binding
MPI_Win_wait(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_WIN_WAIT(WIN, IERROR)
    INTEGER WIN, IERROR
```

Completes an RMA exposure epoch started by a call to MPI_WIN_POST on win. This call matches calls to MPI_WIN_COMPLETE(win) issued by each of the origin processes that were granted access to the window during this epoch. The call to MPI_WIN_WAIT will block until all matching calls to MPI_WIN_COMPLETE have occurred. This guarantees that all these origin processes have completed their RMA accesses to the local window. When the call returns, all these RMA accesses will have completed at the target window.

Figure 12.5 illustrates the use of these four functions. Process 0 puts data in the windows of processes 1 and 2 and process 3 puts data in the window of process 2. Each start call lists the ranks of the processes whose windows will be accessed; each post call lists the ranks of the processes that access the local window. The figure illustrates a possible timing for the events, assuming strong synchronization; in a weak synchronization, the start, put or complete calls may occur ahead of the matching post calls.

Unofficial Draft for Comment Only

```
1
      MPI_WIN_TEST(win, flag)
2
       IN
                                              window object (handle)
                 win
3
       OUT
                 flag
                                              success flag (logical)
4
5
6
      C binding
\overline{7}
      int MPI_Win_test(MPI_Win win, int *flag)
8
     Fortran 2008 binding
9
      MPI_Win_test(win, flag, ierror)
10
          TYPE(MPI_Win), INTENT(IN) :: win
11
          LOGICAL, INTENT(OUT) :: flag
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
      Fortran binding
15
     MPI_WIN_TEST(WIN, FLAG, IERROR)
16
          INTEGER WIN, IERROR
17
          LOGICAL FLAG
18
          This is the nonblocking version of MPI_WIN_WAIT. It returns flag = true if all accesses
19
      to the local window by the group to which it was exposed by the corresponding
20
      MPI_WIN_POST call have been completed as signalled by matching MPI_WIN_COMPLETE
21
      calls, and flag = false otherwise. In the former case MPI_WIN_WAIT would have returned
22
      immediately. The effect of return of MPI_WIN_TEST with flag = true is the same as the
23
      effect of a return of MPI_WIN_WAIT. If flag = false is returned, then the call has no visible
24
      effect.
25
          MPI_WIN_TEST should be invoked only where MPI_WIN_WAIT can be invoked. Once
26
      the call has returned flag = true, it must not be invoked anew, until the window is posted
27
      anew.
28
          Assume that window win is associated with a "hidden" communicator wincomm, used
29
      for communication by the processes of win. The rules for matching of post and start calls
30
      and for matching complete and wait calls can be derived from the rules for matching sends
^{31}
      and receives, by considering the following (partial) model implementation.
32
33
      MPI_WIN_POST(group,0,win) initiates a nonblocking send with tag tag0 to each process
34
          in group, using wincomm. There is no need to wait for the completion of these sends.
35
36
      MPI_WIN_START(group,0,win) initiates a nonblocking receive with tag tag0 from each
37
           process in group, using wincomm. An RMA access to a window in target process i is
38
           delayed until the receive from i is completed.
39
      MPI_WIN_COMPLETE(win) initiates a nonblocking send with tag tag1 to each process
40
           in the group of the preceding start call. No need to wait for the completion of these
41
           sends.
42
43
     MPI_WIN_WAIT(win) initiates a nonblocking receive with tag tag1 from each process in
44
           the group of the preceding post call. Wait for the completion of all receives.
45
46
          No races can occur in a correct program: each of the sends matches a unique receive,
47
     and vice versa.
48
```

Rationale. The design for general active target synchronization requires the user to provide complete information on the communication pattern, at each end of a communication link: each origin specifies a list of targets, and each target specifies a list of origins. This provides maximum flexibility (hence, efficiency) for the implementor: each synchronization can be initiated by either side, since each "knows" the identity of the other. This also provides maximum protection from possible races. On the other hand, the design requires more information than RMA needs: in general, it is sufficient for the origin to know the rank of the target, but not vice versa. Users that want more "anonymous" communication will be required to use the fence or lock mechanisms. (*End of rationale.*)

Advice to users. Assume a communication pattern that is represented by a directed graph $G = \langle V, E \rangle$, where $V = \{0, \ldots, n-1\}$ and $ij \in E$ if origin process i accesses the window at target process j. Then each process i issues a call to MPI_WIN_POST($ingroup_i, \ldots$), followed by a call to MPI_WIN_START($outgroup_i, \ldots$), where $outgroup_i = \{j : ij \in E\}$ and $ingroup_i = \{j : ji \in E\}$. A call is a noop, and can be skipped, if the group argument is empty. After the communications calls, each process that issued a start will issue a complete.

Note that each process may call with a group argument that has different members. (*End of advice to users.*)

Finally, each process that issued a post will issue a wait.

12.5.3 Lock

MDL WINL LOCK (Lash thing would

MPI_WIN_LOCK(lock_type, rank, assert, win)			27
IN	lock_type	either MPI_LOCK_EXCLUSIVE or	28
		MPI_LOCK_SHARED (state)	29
IN	rank	rank of locked window (non-negative integer)	30
IN	assert	program assertion (integer)	31 32
IN	win	window object (handle)	33
			34
C bin	ding		35
int M	PI_Win_lock(int loc	k_type, int rank, int assert, MPI_Win win)	36
			37
	an 2008 binding		38
		rank, assert, win, ierror)	39
II	NTEGER, INTENT(IN)	<pre>:: lock_type, rank, assert</pre>	40
T	YPE(MPI_Win), INTEN	T(IN) :: win	41
II	NTEGER, OPTIONAL, I	NTENT(OUT) :: ierror	42
Fortra	an binding		43
	0	RANK, ASSERT, WIN, IERROR)	44
	-	ANK, ASSERT, WIN, IERROR	45
	-		46
St	arts an RMA access ep	och. The window at the process with rank rank can be accessed	47

Starts an RMA access epoch. The window at the process with rank rank can be accessed 47 by RMA operations on win during that epoch. Multiple RMA access epochs (with calls 48

Unofficial Draft for Comment Only

```
1
     to MPI_WIN_LOCK) can occur simultaneously; however, each access epoch must target a
\mathbf{2}
     different process.
3
4
     MPI_WIN_LOCK_ALL(assert, win)
5
6
       IN
                 assert
                                             program assertion (integer)
7
       IN
                 win
                                             window object (handle)
8
9
     C binding
10
     int MPI_Win_lock_all(int assert, MPI_Win win)
11
12
     Fortran 2008 binding
13
     MPI_Win_lock_all(assert, win, ierror)
14
          INTEGER, INTENT(IN) :: assert
15
          TYPE(MPI_Win), INTENT(IN) :: win
16
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     Fortran binding
18
     MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
19
          INTEGER ASSERT, WIN, IERROR
20
21
          Starts an RMA access epoch to all processes in win, with a lock type of
22
     MPI_LOCK_SHARED. During the epoch, the calling process can access the window memory on
23
     all processes in win by using RMA operations. A window locked with MPI_WIN_LOCK_ALL
^{24}
     must be unlocked with MPI_WIN_UNLOCK_ALL. This routine is not collective — the ALL
25
     refers to a lock on all members of the group of the window.
26
27
           Advice to users. There may be additional overheads associated with using
28
           MPI_WIN_LOCK and MPI_WIN_LOCK_ALL concurrently on the same window. These
29
           overheads could be avoided by specifying the assertion MPI_MODE_NOCHECK when
30
           possible (see Section 12.5.5). (End of advice to users.)
^{31}
32
33
     MPI_WIN_UNLOCK(rank, win)
34
35
       ĪN
                 rank
                                             rank of window (non-negative integer)
36
       IN
                 win
                                             window object (handle)
37
38
     C binding
39
     int MPI_Win_unlock(int rank, MPI_Win win)
40
41
     Fortran 2008 binding
42
     MPI_Win_unlock(rank, win, ierror)
43
          INTEGER, INTENT(IN) :: rank
44
          TYPE(MPI_Win), INTENT(IN) :: win
45
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
     Fortran binding
47
     MPI_WIN_UNLOCK(RANK, WIN, IERROR)
48
```

INTEGER RANK, WIN, IERROR

Completes an RMA access epoch started by a call to MPI_WIN_LOCK on window win. RMA operations issued during this period will have completed both at the origin and at the target when the call returns.

MPI_WIN_UNLOCK_ALL(win)

win

IN

window object (handle)

C binding

int MPI_Win_unlock_all(MPI_Win win)

Fortran 2008 binding

```
MPI_Win_unlock_all(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

MPI_WIN_UNLOCK_ALL(WIN, IERROR) INTEGER WIN, IERROR

Completes a shared RMA access epoch started by a call to MPI_WIN_LOCK_ALL on window win. RMA operations issued during this epoch will have completed both at the origin and at the target when the call returns.

Locks are used to protect accesses to the locked target window effected by RMA calls issued between the lock and unlock calls, and to protect load/store accesses to a locked local or shared memory window executed between the lock and unlock calls. Accesses that are protected by an exclusive lock will not be concurrent at the window site with other accesses to the same window that are lock protected. Accesses that are protected by a shared lock will not be concurrent at the window site with accesses protected by an exclusive lock to the same window.

It is erroneous to have a window locked and exposed (in an exposure epoch) concurrently. For example, a process may not call MPI_WIN_LOCK to lock a target window if the target process has called MPI_WIN_POST and has not yet called MPI_WIN_WAIT; it is erroneous to call MPI_WIN_POST while the local window is locked.

Rationale. An alternative is to require MPI to enforce mutual exclusion between exposure epochs and locking periods. But this would entail additional overheads when locks or active target synchronization do not interact in support of those rare interactions between the two mechanisms. The programming style that we encourage here is that a set of windows is used with only one synchronization mechanism at a time, with shifts from one mechanism to another being rare and involving global synchronization. (*End of rationale.*)

Advice to users. Users need to use explicit synchronization code in order to enforce mutual exclusion between locking periods and exposure epochs on a window. (End of advice to users.)

Unofficial Draft for Comment Only

2

3

4

5 6 7

8

9 10

11

12 13

14

15

16

17

18

19

20 21

22

23

24 25

26

27

28

29

30

 31

32

33

34

35

36 37

38

39

40

41

42

43

44

 $45 \\ 46$

47

1	Implementors may restrict the use	of RMA communication that is synchronized by
2	lock calls to windows in memory alloca	ted by MPI_ALLOC_MEM (Section 9.2),
3	MPI_WIN_ALLOCATE (Section 12.2.2),	MPI_WIN_ALLOCATE_SHARED (Section 12.2.3),
4		ection 12.2.4). Locks can be used portably only in
5	such memory.	
6	Such momory.	
7	Rationale. The implementation	of passive target communication when memory
8		nchronous software agent. Such an agent can be
	· · · ·	chieve better performance, if restricted to specially
9	· · · · · · · · · · · · · · · · · · ·	
10		led altogether if shared memory is used. It seems
11		allows one to use shared memory for third party
12	communication in shared memory	machines.
13	(End of rationale.)	
14		
15	Consider the sequence of calls in the	e example below.
16		
17	Example 12.5 Use of MPI_WIN_LOCK	K and MPI_WIN_UNLOCK.
18	MPI_Win_lock(MPI_LOCK_EXCLUSIVE, r	conk occort win).
19		alik, assert, will),
20	MPI_Put(, rank,, win);	
21	<pre>MPI_Win_unlock(rank, win);</pre>	
22	The call to MPI WIN UNLOCK will	not return until the put transfer has completed at
23		eaves much freedom to implementors. The call to
24		lusive lock on the window is acquired; or, the first
24 25	, and the second s	N_UNLOCK blocks until a lock is acquired — the
		poned until the call to MPI_WIN_UNLOCK occurs.
26		is used to lock a local window, then the call must
27		
28		e lock may protect local load/store accesses to the
29	window issued after the lock call returns	
30		
31	12.5.4 Flush and Sync	
32	All flush and sync functions can be calle	d only within passive target enochs
33	An nush and sync functions can be cane	d only within passive target epochs.
34		
35	MPI_WIN_FLUSH(rank, win)	
36	,	
37	IN rank	rank of target window (non-negative integer)
38	IN win	window object (handle)
39		
40	C binding	
41	int MPI_Win_flush(int rank, MPI_Wi	n win)
42		
43	Fortran 2008 binding	
44	MPI_Win_flush(rank, win, ierror)	
	INTEGER, INTENT(IN) :: rank	
45	TYPE(MPI_Win), INTENT(IN) :: w	vin
46	INTEGER, OPTIONAL, INTENT(OUT)	
47		
48	Fortran binding	

operations.

MPI_WIN_FLUSH(RANK, WIN, IERROR)	1
INTEGER RANK, WIN, IERROR	2
MPI_WIN_FLUSH completes all outstanding RMA operations initiated by the calling	3
process to the target rank on the specified window. The operations are completed both at	4
the origin and at the target.	5
the origin and at the target.	6
	7
MPI_WIN_FLUSH_ALL(win)	8
IN win window object (handle)	9
	10
C binding	11
int MPI_Win_flush_all(MPI_Win win)	12 13
	13
Fortran 2008 binding	15
MPI_Win_flush_all(win, ierror)	16
TYPE(MPI_Win), INTENT(IN) :: win	17
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	18
Fortran binding	19
MPI_WIN_FLUSH_ALL(WIN, IERROR)	20
INTEGER WIN, IERROR	21
All DMA expections issued by the colling process to any target on the specified window	22
All RMA operations issued by the calling process to any target on the specified window prior to this call and in the specified window will have completed both at the origin and at	23
the target when this call returns.	24
the target when this can returns.	25
	26
MPI_WIN_FLUSH_LOCAL(rank, win)	27
IN rank rank of target window (non-negative integer)	28
	29
IN win window object (handle)	30
	31 32
C binding	33
<pre>int MPI_Win_flush_local(int rank, MPI_Win win)</pre>	34
Fortran 2008 binding	35
MPI_Win_flush_local(rank, win, ierror)	36
INTEGER, INTENT(IN) :: rank	37
TYPE(MPI_Win), INTENT(IN) :: win	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
Fortran binding	40
MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)	41
INTEGER RANK, WIN, IERROR	42
	43
Locally completes at the origin all outstanding RMA operations initiated by the calling	44
process to the target process specified by rank on the specified window. For example, after this routine completes, the user may reuse any buffers provided to put, get, or accumulate	45
this routine completes, the user may reuse any buffers provided to put, get, or accumulate operations	46

Unofficial Draft for Comment Only

```
1
     MPI_WIN_FLUSH_LOCAL_ALL(win)
2
       IN
                                             window object (handle)
                 win
3
4
     C binding
5
     int MPI_Win_flush_local_all(MPI_Win win)
6
\overline{7}
     Fortran 2008 binding
8
     MPI_Win_flush_local_all(win, ierror)
9
          TYPE(MPI_Win), INTENT(IN) :: win
10
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
     Fortran binding
12
     MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
13
          INTEGER WIN, IERROR
14
15
          All RMA operations issued to any target prior to this call in this window will have
16
     completed at the origin when MPI_WIN_FLUSH_LOCAL_ALL returns.
17
18
     MPI_WIN_SYNC(win)
19
20
       IN
                                             window object (handle)
                 win
21
22
     C binding
23
     int MPI_Win_sync(MPI_Win win)
^{24}
25
     Fortran 2008 binding
26
     MPI_Win_sync(win, ierror)
          TYPE(MPI_Win), INTENT(IN) :: win
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     Fortran binding
30
     MPI_WIN_SYNC(WIN, IERROR)
^{31}
          INTEGER WIN, IERROR
32
33
          The call MPI_WIN_SYNC synchronizes the private and public window copies of win.
34
     For the purposes of synchronizing the private and public window, MPI_WIN_SYNC has the
35
     effect of ending and reopening an access and exposure epoch on the window (note that it
     does not actually end an epoch or complete any pending MPI RMA operations).
36
37
38
     12.5.5 Assertions
39
     The assert argument in the calls MPI_WIN_POST, MPI_WIN_START, MPI_WIN_FENCE,
40
     MPI_WIN_LOCK, and MPI_WIN_LOCK_ALL is used to provide assertions on the context of
41
     the call that may be used to optimize performance. The assert argument does not change
42
     program semantics if it provides correct information on the program — it is erroneous to
43
     provide incorrect information. Users may always provide assert = 0 to indicate a general
44
     case where no guarantees are made.
45
46
           Advice to users. Many implementations may not take advantage of the information
47
           in assert; some of the information is relevant only for noncoherent shared memory ma-
48
```

chines. Users should consult their implementation's manual to find which information is useful on each system. On the other hand, applications that provide correct assertions whenever applicable are portable and will take advantage of assertion specific optimizations whenever available. (*End of advice to users.*)

Advice to implementors. Implementations can always ignore the assert argument. Implementors should document which assert values are significant on their implementation. (End of advice to implementors.)

assert is the bit vector OR of zero or more of the following integer constants: MPI_MODE_NOCHECK, MPI_MODE_NOSTORE, MPI_MODE_NOPUT, MPI_MODE_NOPRECEDE, and MPI_MODE_NOSUCCEED. The significant options are listed below for each call.

Advice to users. C/C++ users can use bit vector OR(|) to combine these constants; Fortran 90 users can use the bit vector IOR intrinsic. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition!). (End of advice to users.)

MPI_WIN_START:

MPI_MODE_NOCHECK — the matching calls to MPI_WIN_POST have already completed on all target processes when the call to MPI_WIN_START is made. The nocheck option can be specified in a start call if and only if it is specified in each matching post call. This is similar to the optimization of "ready-send" that may save a handshake when the handshake is implicit in the code. However, ready-send is matched by a regular receive, whereas both start and post must specify the nocheck option.

MPI_WIN_POST:

- MPI_MODE_NOCHECK the matching calls to MPI_WIN_START have not yet occurred on any origin processes when the call to MPI_WIN_POST is made. The nocheck option can be specified by a post call if and only if it is specified by each matching start call.
- MPI_MODE_NOSTORE the local window was not updated by stores (or local get or receive calls) since last synchronization. This may avoid the need for cache synchronization at the post call.
- MPI_MODE_NOPUT the local window will not be updated by put or accumulate calls after the post call, until the ensuing (wait) synchronization. This may avoid the need for cache synchronization at the wait call.

MPI_WIN_FENCE:

- MPI_MODE_NOSTORE the local window was not updated by stores (or local get or receive calls) since last synchronization.
- MPI_MODE_NOPUT the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.

47 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

7

8

10

11

12

13 14

15

16

17

18 19

20 21

22

23

 24

25

26

27

28

29 30

 31

32

33

34

35

36

37 38

39

40

41

42 43

44

45

1	$MPI_MODE_NOPRECEDE$ — the fence does not complete any sequence of locally issued
2 3	RMA calls. If this assertion is given by any process in the window group, then it must be given by all processes in the group.
4	MPI_MODE_NOSUCCEED — the fence does not start any sequence of locally issued
5	RMA calls. If the assertion is given by any process in the window group, then it
6 7	must be given by all processes in the group.
8	MPI_WIN_LOCK, MPI_WIN_LOCK_ALL:
9	MPI_MODE_NOCHECK — no other process holds, or will attempt to acquire, a con-
10 11 12 13	flicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.
14	
15 16 17	Advice to users. Note that the nostore and noprecede flags provide information on what happened <i>before</i> the call; the noput and nosucceed flags provide information on what will happen <i>after</i> the call. (<i>End of advice to users.</i>)
18 19	12.5.6 Miscellaneous Clarifications
21 22 23 24 25 26 27	 to that routine. For example, the datatype argument of a MPI_PUT call can be freed as soon as the call returns, even though the communication may not be complete. As in message-passing, datatypes must be committed before they can be used in RMA communication. 12.6 Error Handling
28 29	12.6.1 Error Handlers
30 31 32 33 34 35 36 37	Errors occurring during calls to routines that create MPI windows (e.g., MPI_WIN_CREATE (,comm,)) cause the error handler currently associated with comm to be invoked. All other RMA calls have an input win argument. When an error occurs during such a call, the error handler currently associated with win is invoked. The error handler MPI_ERRORS_ARE_FATAL is associated with win during its creation. Users may change this default by explicitly associating a new error handler with win (see Section 9.3).
38 39	12.6.2 Error Classes
40 41 42 43	The error classes for one-sided communication are defined in Table 12.2. RMA routines may (and almost certainly will) use other MPI error classes, such as MPI_ERR_OP or MPI_ERR_RANK.
44 45	12.7 Semantics and Correctness
46 47	The following rules specify the latest time at which an operation must complete at the origin or the target. The update performed by a get call in the origin process memory is

CHAPTER 12. ONE-SIDED COMMUNICATIONS

⁴⁸ visible when the get operation is complete at the origin (or earlier); the update performed

Unofficial Draft for Comment Only

MPI_ERR_WIN	invalid win argument	1
MPI_ERR_BASE	invalid base argument	2
MPI_ERR_SIZE	invalid size argument	3
MPI_ERR_DISP	invalid disp argument	4
MPI_ERR_LOCKTYPE	invalid locktype argument	5
MPI_ERR_ASSERT	invalid assert argument	6
MPI_ERR_RMA_CONFLICT	conflicting accesses to window	7
MPI_ERR_RMA_SYNC	invalid synchronization of RMA calls	8
MPI_ERR_RMA_RANGE	target memory is not part of the window (in the case	9
	of a window created with	10
	MPI_WIN_CREATE_DYNAMIC, target memory is not	11
	attached)	12
MPI_ERR_RMA_ATTACH	memory cannot be attached (e.g., because of resource	13
	exhaustion)	14
MPI_ERR_RMA_SHARED	memory cannot be shared (e.g., some process in the	15
	group of the specified communicator cannot expose	16
	shared memory)	17
MPI_ERR_RMA_FLAVOR	passed window has the wrong flavor for the called	18
	function	19
		20
		21

Table 12.2: Error classes in one-sided communication routines

by a put or accumulate call in the public copy of the target window is visible when the put or accumulate has completed at the target (or earlier). The rules also specify the latest time at which an update of one window copy becomes visible in another overlapping copy.

- An RMA operation is completed at the origin by the ensuing call to MPI_WIN_COMPLETE, MPI_WIN_FENCE, MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL, MPI_WIN_FLUSH_LOCAL_ALL, MPI_WIN_UNLOCK, or MPI_WIN_UNLOCK_ALL that synchronizes this access at the origin.
- 2. If an RMA operation is completed at the origin by a call to MPI_WIN_FENCE then the operation is completed at the target by the matching call to MPI_WIN_FENCE by the target process.
- 3. If an RMA operation is completed at the origin by a call to MPI_WIN_COMPLETE then the operation is completed at the target by the matching call to MPI_WIN_WAIT by the target process.
- 4. If an RMA operation is completed at the origin by a call to MPI_WIN_UNLOCK, MPI_WIN_UNLOCK_ALL, MPI_WIN_FLUSH(rank=target), or MPI_WIN_FLUSH_ALL, then the operation is completed at the target by that same call.
- 5. An update of a location in a private window copy in process memory becomes visible in the public window copy at latest when an ensuing call to MPI_WIN_POST, MPI_WIN_FENCE, MPI_WIN_UNLOCK, MPI_WIN_UNLOCK_ALL, or MPI_WIN_SYNC is executed on that window by the window owner. In the RMA

Unofficial Draft for Comment Only

22 23

24

25

26 27

28

29

30

 31

32 33

34

35

36

37

38

39 40

41

42

43

44

45 46

47

unified memory model, an update of a location in a private window in process memory becomes visible without additional RMA calls.

6. An update by a put or accumulate call to a public window copy becomes visible in the private copy in process memory at latest when an ensuing call to MPI_WIN_WAIT, MPI_WIN_FENCE, MPI_WIN_LOCK, MPI_WIN_LOCK_ALL, or MPI_WIN_SYNC is executed on that window by the window owner. In the RMA unified memory model, an update by a put or accumulate call to a public window copy eventually becomes visible in the private copy in process memory without additional RMA calls.

10 The MPI_WIN_FENCE or MPI_WIN_WAIT call that completes the transfer from public 11 copy to private copy (6) is the same call that completes the put or accumulate operation in 12the window copy (2, 3). If a put or accumulate access was synchronized with a lock, then 13 the update of the public window copy is complete as soon as the updating process executed 14MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL. In the RMA separate memory model, the 15update of a private copy in the process memory may be delayed until the target process 16executes a synchronization call on that window (6). Thus, updates to process memory can 17always be delayed in the RMA separate memory model until the process executes a suitable 18 synchronization call, while they must complete in the RMA unified model without additional 19synchronization calls. If fence or post-start-complete-wait synchronization is used, updates 20to a public window copy can be delayed in both memory models until the window owner 21executes a synchronization call. When passive target synchronization is used, it is necessary 22 to update the public window copy even if the window owner does not execute any related 23synchronization call. 24

The rules above also define, by implication, when an update to a public window copy becomes visible in another overlapping public window copy. Consider, for example, two overlapping windows, win1 and win2. A call to MPI_WIN_FENCE(0, win1) by the window owner makes visible in the process memory previous updates to window win1 by remote processes. A subsequent call to MPI_WIN_FENCE(0, win2) makes these updates visible in the public copy of win2.

The behavior of some MPI RMA operations may be *undefined* in certain situations. For 31 example, the result of several origin processes performing concurrent MPI_PUT operations 32 to the same target location is undefined. In addition, the result of a single origin process 33 performing multiple MPI_PUT operations to the same target location within the same 34access epoch is also undefined. The result at the target may have all of the data from one 35 of the MPI_PUT operations (the "last" one, in some sense), bytes from some of each of 36 the operations, or something else. In MPI-2, such operations were erroneous. That meant 37 that an MPI implementation was permitted to raise an error. Thus, user programs or tools 38 that used MPI RMA could not portably permit such operations, even if the application code 39 could function correctly with such an undefined result. In MPI-3, these operations are not 40 erroneous, but do not have a defined behavior. 41

Rationale. As discussed in [7], requiring operations such as overlapping puts to
 be erroneous makes it difficult to use MPI RMA to implement programming models—
 such as Unified Parallel C (UPC) or SHMEM—that permit these operations. Further,
 while MPI-2 defined these operations as erroneous, the MPI Forum is unaware of any
 implementation that enforces this rule, as it would require significant overhead. Thus,
 relaxing this condition does not impact existing implementations or applications. (End
 of rationale.)

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

Advice to implementors. Overlapping accesses are undefined. However, to assist users in debugging code, implementations may wish to provide a mode in which such operations are detected and reported to the user. Note, however, that in MPI-3, such operations must not raise an error. (*End of advice to implementors.*)

A program with a well-defined outcome in the MPI_WIN_SEPARATE memory model must obey the following rules.

- S1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update becomes visible in the private window copy in process memory.
- S2. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started, until the update becomes visible in the public window copy. There is one exception to this rule, in the case where the same variable is updated by two concurrent accumulates with the same predefined datatype, on the same window. Additional restrictions on the operation apply, see the info key accumulate_ops in Section 12.2.1.
- S3. A put or accumulate must not access a target window once a store or a put or accumulate update to another (overlapping) target window has started on a location in the target window, until the update becomes visible in the public copy of the window. Conversely, a store to process memory to a location in a window must not start once a put or accumulate update to that target window has started, until the put or accumulate update becomes visible in process memory. In both cases, the restriction applies to operations even if they access disjoint locations in the window.

Rationale. The last constraint on correct RMA accesses may seem unduly restrictive, as it forbids concurrent accesses to nonoverlapping locations in a window. The reason for this constraint is that, on some architectures, explicit coherence restoring operations may be needed at synchronization points. A different operation may be needed for locations that were updated by stores and for locations that were remotely updated by put or accumulate operations. Without this constraint, the MPI library would have to track precisely which locations in a window were updated by a put or accumulate call. The additional overhead of maintaining such information is considered prohibitive. (*End of rationale.*)

Note that MPI_WIN_SYNC may be used within a passive target epoch to synchronize the private and public window copies (that is, updates to one are made visible to the other).

In the MPI_WIN_UNIFIED memory model, the rules are simpler because the public and private windows are the same. However, there are restrictions to avoid concurrent access to the same memory locations by different processes. The rules that a program with a well-defined outcome must obey in this case are:

- U1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update is complete, subject to the following special case.
- U2. Accessing a location in the window that is also the target of a remote update is valid (not erroneous) but the precise result will depend on the behavior of the implementation. Updates from a remote process will appear in the memory of the target, but 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

7 8

9

10

11

12

13

14

15

16

17 18

19

20

21

22

23

 24

25 26

27

28

29

30

 31

32

33

34

35

36

37

38

39

40

41 42

43

44

there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory of the target, the data remains until replaced by another update. This permits polling on a location for a change from zero to non-zero or for a particular value, but not polling and comparing the relative magnitude of values. Users are cautioned that polling on one memory location and then accessing a different memory location has defined behavior only if the other rules given here and in this chapter are followed.

Advice to users. Some compiler optimizations can result in code that maintains the sequential semantics of the program, but violates this rule by introducing temporary values into locations in memory. Most compilers only apply such transformations under very high levels of optimization and users should be aware that such aggressive optimization may produce unexpected results. (End of advice to users.)

- U3. Updating a location in the window with a store operation that is also the target 16of a remote read (but not update) is valid (not erroneous) but the precise result 17 will depend on the behavior of the implementation. Store updates will appear in 18 memory, but there are no atomicity or ordering guarantees if more than one byte is 19 updated. Updates are stable in the sense that once data appears in memory, the data 20remains until replaced by another update. This permits updates to memory with 21store operations without requiring an RMA epoch. Users are cautioned that remote 22 accesses to a window that is updated by the local process has defined behavior only 23if the other rules given here and elsewhere in this chapter are followed. 24
 - U4. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started and until the update completes at the target. There is one exception to this rule: in the case where the same location is updated by two concurrent accumulates with the same predefined datatype on the same window. Additional restrictions on the operation apply; see the info key accumulate_ops in Section 12.2.1.
 - U5. A put or accumulate must not access a target window once a store, put, or accumulate update to another (overlapping) target window has started on the same location in the target window and until the update completes at the target window. Conversely, a store operation to a location in a window must not start once a put or accumulate update to the same location in that target window has started and until the put or accumulate update completes at the target.

Advice to users. In the unified memory model, in the case where the window is in shared memory, MPI_WIN_SYNC can be used to order store operations and make store updates to the window visible to other processes and threads. Use of this routine is necessary to ensure portable behavior when point-to-point, collective, or shared memory synchronization is used in place of an RMA synchronization routine. MPI_WIN_SYNC should be called by the writer before the non-RMA synchronization operation and by the reader after the non-RMA synchronization, as shown in Example 12.21. (End of advice to users.)

⁴⁸ A program that violates these rules has undefined behavior.

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10 11

12

13

14 15

25

26

27

28

29

30

31 32

33

34

35

36

37

38 39

40

41

42

43 44

45

46

Advice to users. A user can write correct programs by following the following rules:

- fence: During each period between fence calls, each window is either updated by put or accumulate calls, or updated by stores, but not both. Locations updated by put or accumulate calls should not be accessed during the same period (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated during the same period.
- **post-start-complete-wait:** A window should not be updated with store operations while posted if it is being updated by put or accumulate calls. Locations updated by put or accumulate calls should not be accessed while the window is posted (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated while the window is posted.

With the post-start synchronization, the target process can tell the origin process that its window is now ready for RMA access; with the complete-wait synchronization, the origin process can tell the target process that it has finished its RMA accesses to the window.

- **lock:** Updates to the window are protected by exclusive locks if they may conflict. Nonconflicting accesses (such as read-only accesses or accumulate accesses) are protected by shared locks, both for load/store accesses and for RMA accesses.
- changing window or synchronization mode: One can change synchronization mode, or change the window used to access a location that belongs to two over-lapping windows, when the process memory and the window copy are guaranteed to have the same values. This is true after a local call to MPI_WIN_FENCE, if RMA accesses to the window are synchronized with fences; after a local call to MPI_WIN_WAIT, if the accesses are synchronized with post-start-complete-wait; after the call at the origin (local or remote) to MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL if the accesses are synchronized with locks.

In addition, a process should not access the local buffer of a get operation until the operation is complete, and should not update the local buffer of a put or accumulate operation until that operation is complete.

The RMA synchronization operations define when updates are guaranteed to become visible in public and private windows. Updates may become visible earlier, but such behavior is implementation dependent. (*End of advice to users.*)

The semantics are illustrated by the following examples:

Example 12.6 The following example demonstrates updating a memory location inside a window for the separate memory model, according to Rule 5. The MPI_WIN_LOCK and MPI_WIN_UNLOCK calls around the store to X in process B are necessary to ensure consistency between the public and private copies of the window.

 $\mathbf{2}$

 24

 31

```
1
     Process A:
                                    Process B:
\mathbf{2}
                                    window location X
3
4
                                    MPI_Win_lock(EXCLUSIVE, B)
5
                                    store X /* local update to private copy of B */
6
                                    MPI_Win_unlock(B)
7
                                    /* now visible in public window copy */
8
9
     MPI_Barrier
                                    MPI_Barrier
10
11
     MPI_Win_lock(EXCLUSIVE, B)
12
     MPI_Get(X) /* ok, read from public window */
13
     MPI_Win_unlock(B)
14
15
     Example 12.7 In the RMA unified model, although the public and private copies of the
16
     windows are synchronized, caution must be used when combining load/stores and multi-
17
     process synchronization. Although the following example appears correct, the compiler or
18
     hardware may delay the store to X after the barrier, possibly resulting in the MPI_GET
19
     returning an incorrect value of X.
20
21
     Process A:
                               Process B:
22
                               window location X
23
^{24}
                               store X /* update to private & public copy of B */
25
     MPI_Barrier
                               MPI_Barrier
26
     MPI_Win_lock_all
27
     MPI_Get(X) /* ok, read from window */
28
     MPI_Win_flush_local(B)
29
     /* read value in X */
30
     MPI_Win_unlock_all
31
32
     MPI_BARRIER provides process synchronization, but not memory synchronization. The
33
     example could potentially be made safe through the use of compiler- and hardware-specific
34
     notations to ensure the store to X occurs before process B enters the MPI_BARRIER. The
35
     use of one-sided synchronization calls, as shown in Example 12.6, also ensures the correct
36
     result.
37
38
     Example 12.8 The following example demonstrates the reading of a memory location
39
     updated by a remote process (Rule 6) in the RMA separate memory model. Although
40
     the MPI_WIN_UNLOCK on process A and the MPI_BARRIER ensure that the public copy
41
     on process B reflects the updated value of X, the call to MPI_WIN_LOCK by process B is
42
     necessary to synchronize the private copy with the public copy.
43
                                    Process B:
     Process A:
44
                                    window location X
45
46
     MPI_Win_lock(EXCLUSIVE, B)
47
     MPI_Put(X) /* update to public window */
48
```

MPI_Win_unlock(B)		1
		2
MPI_Barrier	MPI_Barrier	3
		4
	MPI_Win_lock(EXCLUSIVE, B)	5
	<pre>/* now visible in private copy of B */</pre>	6
	load X	7
	MPI_Win_unlock(B)	8
NT ((1 ((1 ())))))))))))))		9
- /	he barrier is not critical to the semantic correctness. The	10
0	ees a remote process will not modify the public copy after	11
MPI_WIN_LOCK synchronizes	the private and public copies. A polling implementation	12
looking for changes in X on pro	cess B would be semantically correct. The barrier is required	13
to ensure that process A perfo	rms the put operation before process B performs the load of	14
Χ.		15
		16
_	mple 12.7, the following example is unsafe even in the unified	17
,	In not be guaranteed to occur after the MPI_BARRIER. While	18
D D 1 . 1.		

m Process B does not need to explicitly synchronize the public and private copies through MPI_WIN_LOCK as the MPI_PUT will update both the public and private copies of the window, the scheduling of the load could result in old values of X being returned. Compiler and hardware specific notations could ensure the load occurs after the data is updated, or explicit one-sided synchronization calls can be used to ensure the proper result.

Process A:	Process B:	24
TIOCESS A.		25
	window location X	26
MPI_Win_lock_all		20
	ndour */	27
<pre>MPI_Put(X) /* update to wi</pre>	ndow */	28
MPI_Win_flush(B)		29
MDT Demoisur	NDT Down's	30
MPI_Barrier	MPI_Barrier	31
	load X	
MPI_Win_unlock_all		32
MF1_WIN_UNIOCK_all		33
	7	34

Example 12.10 The following example further clarifies Rule 5. MPI_WIN_LOCK and MPI_WIN_LOCK_ALL do not update the public copy of a window with changes to the private copy. Therefore, there is no guarantee that process A in the following sequence will see the value of X as updated by the local store by process B before the lock.

		39
Process A:	Process B:	40
	window location X	41
		42
	store X /* update to private copy of B */	43
	MPI_Win_lock(SHARED, B)	44
MPI_Barrier	MPI_Barrier	45
		46
MPI_Win_lock(SHARED, B)		47
MPI_Get(X) /* X may be the	X before the store */	48

19

20

21

22

3536

37

1 MPI_Win_unlock(B) $\mathbf{2}$ MPI_Win_unlock(B) 3 /* update on X now visible in public window */ 4 The addition of an MPI_WIN_SYNC before the call to MPI_BARRIER by process B would 5guarantee process A would see the updated value of X, as the public copy of the window 6 would be explicitly synchronized with the private copy. 7 8 **Example 12.11** Similar to the previous example, Rule 5 can have unexpected implications 9 for general active target synchronization with the RMA separate memory model. It is not 10 guaranteed that process B reads the value of **X** as per the local update by process A, because 11 neither MPI_WIN_WAIT nor MPI_WIN_COMPLETE calls by process A ensure visibility in 12the public window copy. 13 14Process A: Process B: 15window location X 16window location Y 1718 store Y 19 MPI_Win_post(A, B) /* Y visible in public window * 20MPI_Win_start(A) MPI_Win_start(A) 2122store X /* update to private window */ 23 24 MPI_Win_complete MPI_Win_complete 25MPI_Win_wait 26/* update on X may not yet visible in public window */ 2728MPI_Barrier MPI_Barrier 29 30 MPI_Win_lock(EXCLUSIVE, A) 31MPI_Get(X) /* may return an obsolete value */ 32 MPI_Get(Y) 33 MPI_Win_unlock(A) 34 35 To allow process B to read the value of X stored by A the local store must be replaced by 36 a local MPI_PUT that updates the public window copy. Note that by this replacement X 37 may become visible in the private copy of process A only after the MPI_WIN_WAIT call in 38 process A. The update to Y made before the MPI_WIN_POST call is visible in the public 39 window after the MPI_WIN_POST call and therefore process B will read the proper value 40of Y. The MPI_GET(Y) call could be moved to the epoch started by the MPI_WIN_START 41 operation, and process B would still get the value stored by process A. 42**Example 12.12** The following example demonstrates the interaction of general active 43 target synchronization with local read operations with the RMA separate memory model. 44Rules 5 and 6 do not guarantee that the private copy of X at process B has been updated 45before the load takes place. 464748

Process A:

Process B: window location X

```
MPI_Win_lock(EXCLUSIVE, B)
MPI_Put(X) /* update to public window */
MPI_Win_unlock(B)
```

MPI_Barrier MPI_Barrier

MPI_Win_complete MPI_Win_wait

To ensure that the value put by process A is read, the local load must be replaced with a local MPI_GET operation, or must be placed after the call to MPI_WIN_WAIT.

12.7.1 Atomicity

The outcome of concurrent accumulate operations to the same location with the same predefined datatype is as if the accumulates were done at that location in some serial order. Additional restrictions on the operation apply; see the info key accumulate_ops in Section 12.2.1. Concurrent accumulate operations with different origin and target pairs are not ordered. Thus, there is no guarantee that the entire call to an accumulate operation is executed atomically. The effect of this lack of atomicity is limited: The previous correctness conditions imply that a location updated by a call to an accumulate operation has completed (at the target). Different interleavings can lead to different results only to the extent that computer arithmetics are not truly associative or commutative. The outcome of accumulate operations with overlapping types of different sizes or target displacements is undefined.

12.7.2 Ordering

Accumulate calls enable element-wise atomic read and write to remote memory locations. MPI specifies ordering between accumulate operations from an origin process to the same (or overlapping) memory locations at a target process on a per-datatype granularity. The default ordering is strict ordering, which guarantees that overlapping updates from the same origin to a remote location are committed in program order and that reads (e.g., with MPI_GET_ACCUMULATE) and writes (e.g., with MPI_ACCUMULATE) are executed and committed in program order. Ordering only applies to operations originating at the same origin that access overlapping target memory regions. MPI does not provide any guarantees for accesses or updates from different origin processes to overlapping target memory regions.

The default strict ordering may incur a significant performance penalty. MPI specifies the info key "accumulate_ordering" to allow relaxation of the ordering semantics when specified 48

Unofficial Draft for Comment Only

 24

 $45 \\ 46$

1 to any window creation function. The values for this key are as follows. If set to "none", $\mathbf{2}$ then no ordering will be guaranteed for accumulate calls. This was the behavior for RMA 3 in MPI-2 but is not the default in MPI-3. The key can be set to a comma-separated list 4 of required access orderings at the target. Allowed values in the comma-separated list are $\mathbf{5}$ "rar", "war", "raw", and "waw" for read-after-read, write-after-read, read-after-write, and 6 write-after-write ordering, respectively. These indicate whether operations of the specified $\overline{7}$ type complete in the order they were issued. For example, "raw" means that any writes must 8 complete at the target before subsequent reads. These ordering requirements apply only to 9 operations issued by the same origin process and targeting the same target process. The 10 default value for "accumulate_ordering" is rar, raw, war, waw, which implies that writes complete 11at the target in the order in which they were issued, reads complete at the target before any 12writes that are issued after the reads, and writes complete at the target before any reads 13that are issued after the writes. Any subset of these four orderings can be specified. For 14example, if only read-after-read and write-after-write ordering is required, then the value of 15the "accumulate_ordering" key could be set to rar, waw. The order of values is not significant. 16Note that the above ordering semantics apply only to accumulate operations, not put

¹⁷ and get. Put and get within an epoch are unordered.

18 19

20

12.7.3 Progress

One-sided communication has the same progress requirements as point-to-point communication: once a communication is enabled it is guaranteed to complete. RMA calls must have local semantics, except when required for synchronization with other RMA calls.

There is some fuzziness in the definition of the time when a RMA communication 24 becomes enabled. This fuzziness provides to the implementor more flexibility than with 25point-to-point communication. Access to a target window becomes enabled once the corre-26sponding synchronization (such as MPI_WIN_FENCE or MPI_WIN_POST) has executed. On 27the origin process, an RMA communication may become enabled as soon as the correspond-28ing put, get or accumulate call has executed, or as late as when the ensuing synchronization 29 call is issued. Once the communication is enabled both at the origin and at the target, the 30 communication must complete. 31

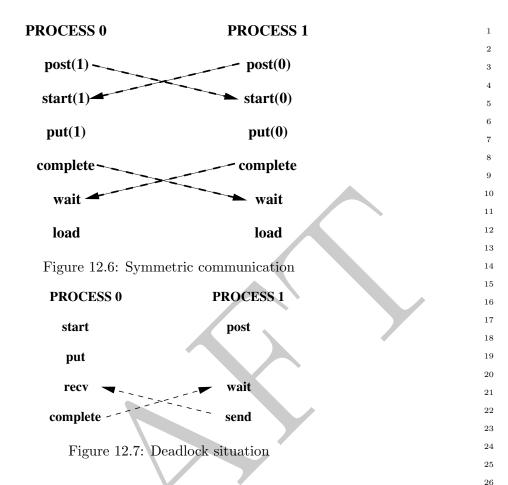
Consider the code fragment in Example 12.4. Some of the calls may block if the target window is not posted. However, if the target window is posted, then the code fragment must complete. The data transfer may start as soon as the put call occurs, but may be delayed until the ensuing complete call occurs.

Consider the code fragment in Example 12.5. Some of the calls may block if another process holds a conflicting lock. However, if no conflicting lock is held, then the code fragment must complete.

Consider the code illustrated in Figure 12.6. Each process updates the window of the other process using a put operation, then accesses its own window. The post calls are nonblocking, and should complete. Once the post calls occur, RMA access to the windows is enabled, so that each process should complete the sequence of calls start-put-complete. Once these are done, the wait calls should complete at both processes. Thus, this communication should not deadlock, irrespective of the amount of data transferred.

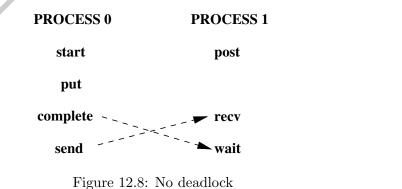
Assume, in the last example, that the order of the post and start calls is reversed at each process. Then, the code may deadlock, as each process may block on the start call, waiting for the matching post to occur. Similarly, the program will deadlock if the order of the complete and wait calls is reversed at each process.

Unofficial Draft for Comment Only



The following two examples illustrate the fact that the synchronization between complete and wait is not symmetric: the wait call blocks until the complete executes, but not vice versa. Consider the code illustrated in Figure 12.7. This code will deadlock: the wait of process 1 blocks until process 0 calls complete, and the receive of process 0 blocks until process 1 calls send. Consider, on the other hand, the code illustrated in Figure 12.8. This code will not deadlock. Once process 1 calls post, then the sequence start, put, complete on process 0 can proceed to completion. Process 0 will reach the send call, allowing the receive call of process 1 to complete.

Rationale. MPI implementations must guarantee that a process makes progress on all enabled communications it participates in, while blocked on an MPI call. This is true



 $46 \\ 47$

for send-receive communication and applies to RMA communication as well. Thus, in the example in Figure 12.8, the put and complete calls of process 0 should complete while process 1 is blocked on the receive call. This may require the involvement of process 1, e.g., to transfer the data put, while it is blocked on the receive call.

A similar issue is whether such progress must occur while a process is busy computing, or blocked in a non-MPI call. Suppose that in the last example the send-receive pair is replaced by a write-to-socket/read-from-socket pair. Then MPI does not specify whether deadlock is avoided. Suppose that the blocking receive of process 1 is replaced by a very long compute loop. Then, according to one interpretation of the MPI standard, process 0 must return from the complete call after a bounded delay, even if process 1 does not reach any MPI call in this period of time. According to another interpretation, the complete call may block until process 1 reaches the wait call, or reaches another MPI call. The qualitative behavior is the same, under both interpretations, unless a process is caught in an infinite compute loop, in which case the difference may not matter. However, the quantitative expectations are different. Different MPI implementations reflect these different interpretations. While this ambiguity is unfortunate, the MPI Forum decided not to define which interpretation of the standard is the correct one, since the issue is contentious. (*End of rationale*.)

12.7.4 Registers and Compiler Optimizations

Advice to users. All the material in this section is an advice to users. (End of advice to users.)

A coherence problem exists between variables kept in registers and the memory values of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A get will not return the latest variable value, and a put may be overwritten when the register is stored back in memory. Note that these issues are unrelated to the RMA memory model; that is, these issues apply even if the memory model is MPI_WIN_UNIFIED.

The problem is illustrated by the following code:

32			
33	Source of Process 1	Source of Process 2	Executed in Process 2
34	bbbb = 777	buff = 999	reg_A:=999
35	call MPI_WIN_FENCE	call MPI_WIN_FENCE	
36	call MPI_PUT(bbbb		stop appl.thread
37	into buff of process 2)		buff:=777 in PUT handler
38			continue appl.thread
39	call MPI_WIN_FENCE	call MPI_WIN_FENCE	
40		ccc = buff	ccc:=reg_A
41			

In this example, variable **buff** is allocated in the register **reg_A** and therefore **ccc** will have the old value of **buff** and not the new value 777.

This problem, which also afflicts in some cases send/receive communication, is discussed more at length in Section 19.1.16.

Programs written in C avoid this problem, because of the semantics of C. Many Fortran
 compilers will avoid this problem, without disabling compiler optimizations. However, in
 order to avoid register coherence problems in a completely portable manner, users should

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19 20

21 22

23

24

 31

42

restrict their use of RMA windows to variables stored in modules or COMMON blocks. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 19.1.10-19.1.20. Sections 19.1.17 to 19.1.17discuss several solutions for the problem in this example.

12.8 Examples

Example 12.13 The following example shows a generic loosely synchronous, iterative code, using fence synchronization. The window at each process consists of array A, which contains the origin and target buffers of the put calls.

```
. . .
while (!converged(A)) {
 update(A);
 MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
 for(i=0; i < toneighbors; i++)</pre>
    MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                          todisp[i], 1, totype[i], win);
 MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
}
```

The same code could be written with get rather than put. Note that, during the communication phase, each window is concurrently read (as origin buffer of puts) and written (as target buffer of puts). This is OK, provided that there is no overlap between the target buffer of a put and another communication buffer.

Example 12.14 Same generic example, with more computation/communication overlap. We assume that the update phase is broken into two subphases: the first, where the "boundary," which is involved in communication, is updated, and the second, where the "core," which neither uses nor provides communicated data, is updated.

```
while (!converged(A)) {
 update_boundary(A);
 MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
  for(i=0; i < fromneighbors; i++)</pre>
    MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
                    fromdisp[i], 1, fromtype[i], win);
 update_core(A);
  MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
}
```

The get communication can be concurrent with the core update, since they do not access the 43 44same locations, and the local update of the origin buffer by the get call can be concurrent with the local update of the core by the update_core call. In order to get similar overlap with put communication we would need to use separate windows for the core and for the boundary. This is required because we do not allow local stores to be concurrent with puts on the same, or on overlapping, windows.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4 56

> 7 8 9

10

11

1213

14

15

16

17

18

19

20

2122

23

 24

25

2627

28

29

30

31 32 33

34

35

36

37

38

39

40

41

42

4546

47

```
1
     Example 12.15 Same code as in Example 12.13, rewritten using post-start-complete-wait.
\mathbf{2}
3
     while (!converged(A)) {
4
       update(A);
5
       MPI_Win_post(fromgroup, 0, win);
6
       MPI_Win_start(togroup, 0, win);
7
       for(i=0; i < toneighbors; i++)</pre>
8
         MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
9
                  todisp[i], 1, totype[i], win);
10
       MPI_Win_complete(win);
11
       MPI_Win_wait(win);
12
     }
13
14
15
     Example 12.16 Same example, with split phases, as in Example 12.14.
16
17
     . . .
     while (!converged(A)) {
18
       update_boundary(A);
19
       MPI_Win_post(togroup, MPI_MODE_NOPUT, win);
20
       MPI_Win_start(fromgroup, 0, win);
21
       for(i=0; i < fromneighbors; i++)</pre>
22
         MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
23
                  fromdisp[i], 1, fromtype[i], win);
24
       update_core(A);
25
       MPI_Win_complete(win);
26
       MPI_Win_wait(win);
27
     }
28
29
30
     Example 12.17 A checkerboard, or double buffer communication pattern, that allows
^{31}
     more computation/communication overlap. Array A0 is updated using values of array A1,
32
     and vice versa. We assume that communication is symmetric: if process A gets data from
33
     process B, then process B gets data from process A. Window wini consists of array Ai.
34
35
36
     if (!converged(A0,A1))
37
       MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
     MPI_Barrier(comm0);
38
     /* the barrier is needed because the start call inside the
39
40
     loop uses the nocheck option */
41
     while (!converged(A0, A1)) {
42
       /* communication on AO and computation on A1 */
       update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
43
       MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
44
       for(i=0; i < fromneighbors; i++)</pre>
45
         MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
46
47
                      fromdisp0[i], 1, fromtype0[i], win0);
48
       update1(A1); /* local update of A1 that is
```

Unofficial Draft for Comment Only

```
concurrent with communication that updates A0 */
MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
MPI_Win_complete(win0);
MPI_Win_wait(win0);
/* communication on A1 and computation on A0 */
update2(A0, A1); /* local update of A0 that depends on A1 (and A0) */
MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win1);
for(i=0; i < fromneighbors; i++)
MPI_Get(&tobuf1[i], 1, totype1[i], neighbor[i],
fromdisp1[i], 1, fromtype1[i], win1);
update1(A0); /* local update of A0 that depends on A0 only,
concurrent with communication that updates A1 */
```

```
if (!converged(A0,A1))
```

```
MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
MPI_Win_complete(win1);
MPI_Win_wait(win1);
```

```
}
```

A process posts the local window associated with win0 before it completes RMA accesses to the remote windows associated with win1. When the wait(win1) call returns, then all neighbors of the calling process have posted the windows associated with win0. Conversely, when the wait(win0) call returns, then all neighbors of the calling process have posted the windows associated with win1. Therefore, the nocheck option can be used with the calls to MPI_WIN_START.

Put calls can be used, instead of get calls, if the area of array AO (resp. A1) used by the update(A1, AO) (resp. update(AO, A1)) call is disjoint from the area modified by the RMA communication. On some systems, a put call may be more efficient than a get call, as it requires information exchange only in one direction.

In the next several examples, for conciseness, the expression

z = MPI_Get_accumulate(...)

means to perform an MPI_GET_ACCUMULATE with the result buffer (given by result_addr in the description of MPI_GET_ACCUMULATE) on the left side of the assignment, in this case, z. This format is also used with MPI_COMPARE_AND_SWAP and MPI_COMM_SIZE. Process B... refers to any process other than A.

Example 12.18 The following example implements a naive, non-scalable counting semaphore. The example demonstrates the use of MPI_WIN_SYNC to manipulate the public copy of X, as well as MPI_WIN_FLUSH to complete operations without ending the access epoch opened with MPI_WIN_LOCK_ALL. To avoid the rules regarding synchronization of the public and private copies of windows, MPI_ACCUMULATE and MPI_GET_ACCUMULATE are used to write to or read from the local public copy.

Process A:	Process B:
MPI_Win_lock_all	MPI_Win_lock_all
window location X	
X=MPI_Comm_size()	
MPI_Win_sync	

1

 $\mathbf{2}$

3

4 5 6

7

9 10

11

12

13 14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30 31

32

33

34

35

36 37

38

39

40

41

42

```
1
     MPI_Barrier
                                                    MPI_Barrier
\mathbf{2}
3
     MPI_Accumulate(X, MPI_SUM, -1)
                                                    MPI_Accumulate(X, MPI_SUM, -1)
4
\mathbf{5}
     stack variable z
                                                    stack variable z
6
     do
                                                    do
7
       z = MPI_Get_accumulate(X,
                                                      z = MPI_Get_accumulate(X,
8
                                                            MPI_NO_OP, 0)
             MPI_NO_OP, 0)
9
       MPI_Win_flush(A)
                                                      MPI_Win_flush(A)
10
     while(z!=0)
                                                    while(z!=0)
11
12
     MPI_Win_unlock_all
                                                    MPI_Win_unlock_all
13
14
     Example 12.19 Implementing a critical region between two processes (Peterson's al-
15
     gorithm). Despite their appearance in the following example, MPI_WIN_LOCK_ALL and
16
     MPI_WIN_UNLOCK_ALL are not collective calls, but it is frequently useful to start shared
17
     access epochs to all processes from all other processes in a window. Once the access epochs
18
     are established, accumulate communication operations and flush and sync synchronization
19
     operations can be used to read from or write to the public copy of the window.
20
21
     Process A:
                                               Process B:
22
     window location X
                                                window location Y
23
     window location T
^{24}
25
                                                MPI_Win_lock_all
     MPI_Win_lock_all
26
     X=1
                                                Y=1
27
                                                MPI_Win_sync
     MPI_Win_sync
28
     MPI_Barrier
                                               MPI_Barrier
29
     MPI_Accumulate(T, MPI_REPLACE, 1)
                                               MPI_Accumulate(T, MPI_REPLACE, 0)
30
     stack variables t,y
                                                stack variable t,x
^{31}
     t=1
                                                t=0
32
     y=MPI_Get_accumulate(Y,
                                                x=MPI_Get_accumulate(X,
33
        MPI_NO_OP, 0)
                                                   MPI_NO_OP, 0)
34
     while(y==1 && t==1) do
                                                while(x==1 && t==0) do
35
       y=MPI_Get_accumulate(Y,
                                                  x=MPI_Get_accumulate(X,
36
          MPI_NO_OP, 0)
                                                     MPI_NO_OP, 0)
37
       t=MPI_Get_accumulate(T,
                                                  t=MPI_Get_accumulate(T,
38
          MPI_NO_OP, 0)
                                                     MPI_NO_OP, 0)
39
       MPI_Win_flush_all
                                                  MPI_Win_flush(A)
40
     done
                                                done
41
     // critical region
                                                // critical region
42
     MPI_Accumulate(X, MPI_REPLACE, 0)
                                               MPI_Accumulate(Y, MPI_REPLACE, 0)
43
     MPI_Win_unlock_all
                                               MPI_Win_unlock_all
44
45
     Example 12.20 Implementing a critical region between multiple processes with compare
46
```

and swap. The call to MPI_WIN_SYNC is necessary on Process A after local initialization of A to guarantee the public copy has been updated with the initialization value found in

Unofficial Draft for Comment Only

the private copy. It would also be valid to call MPI_ACCUMULATE with MPI_REPLACE to directly initialize the public copy. A call to MPI_WIN_FLUSH would be necessary to assure A in the public copy of Process A had been updated before the barrier.

Process A: MPI_Win_lock_all atomic location A	Process B: MPI_Win_lock_all
A=0	
MPI_Win_sync	
MPI_Barrier	MPI_Barrier
stack variable r=1	stack variable r=1
while(r != 0) do	while(r != 0) do
r = MPI_Compare_and_swap(A, 0, 1) MPI_Win_flush(A)	<pre>r = MPI_Compare_and_swap(A, 0, 1) MPI_Win_flush(A)</pre>
done	done
// critical region	// critical region
<pre>r = MPI_Compare_and_swap(A, 1, 0)</pre>	<pre>r = MPI_Compare_and_swap(A, 1, 0)</pre>
MPI_Win_unlock_all	MPI_Win_unlock_all

Example 12.21 The following example demonstrates the proper synchronization in the unified memory model when a data transfer is implemented with load and store in the case of windows in shared memory (instead of MPI_PUT or MPI_GET) and the synchronization between processes is performed using point-to-point communication. The synchronization between processes must be supplemented with a memory synchronization through calls to MPI_WIN_SYNC, which act locally as a processor-memory barrier. In Fortran, if MPI_ASYNC_PROTECTS_NONBLOCKING is .FALSE. or the variable X is not declared as ASYNCHRONOUS, reordering of the accesses to the variable X must be prevented with MPI_F_SYNC_REG operations. (No equivalent function is needed in C.)

The variable X is contained within a shared memory window and X corresponds to the same memory location at both processes. The MPI_WIN_SYNC operation performed by process A ensures completion of the load/store operations issued by process A. The MPI_WIN_SYNC operation performed by process B ensures that process A's updates to X are visible to process B.

Process A:	Process B:	35
MPI_WIN_LOCK_ALL(MPI_WIN_LOCK_ALL(36
MPI_MODE_NOCHECK,win)	MPI_MODE_NOCHECK,win)	37
		38
DO	DO	39
X=		40
		41
MPI_F_SYNC_REG(X)		42
MPI_WIN_SYNC(win)		43
MPI_SEND	MPI_RECV	44
	MPI_WIN_SYNC(win)	45
	MPI_F_SYNC_REG(X)	46
		47
	print X	48

 24

Unofficial Draft for Comment Only

```
1
\mathbf{2}
                                              MPI_F_SYNC_REG(X)
3
       MPI_RECV
                                              MPI_SEND
4
       MPI_F_SYNC_REG(X)
\mathbf{5}
     END DO
                                           END DO
6
7
     MPI_WIN_UNLOCK_ALL(win)
                                           MPI_WIN_UNLOCK_ALL(win)
8
9
     Example 12.22 The following example shows how request-based operations can be used
10
     to overlap communication with computation. Each process fetches, processes, and writes
11
     the result for NSTEPS chunks of data. Instead of a single buffer, M local buffers are used to
12
     allow up to M communication operations to overlap with computation.
13
14
     int
                   i, j;
15
     MPI_Win
                   win;
16
     MPI_Request put_req[M] = { MPI_REQUEST_NULL };
17
     MPI_Request get_req;
18
     double
                   *baseptr;
19
     double
                   data[M][N];
20
21
     MPI_Win_allocate(NSTEPS*N*sizeof(double), sizeof(double), MPI_INFO_NULL,
22
       MPI_COMM_WORLD, &baseptr, &win);
23
^{24}
     MPI_Win_lock_all(0, win);
25
26
     for (i = 0; i < NSTEPS; i++) {</pre>
27
      if (i<M)
28
         j=i;
29
       else
30
         MPI_Waitany(M, put_req, &j, MPI_STATUS_IGNORE);
^{31}
32
      MPI_Rget(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
33
                 &get_req);
34
      MPI_Wait(&get_req,MPI_STATUS_IGNORE);
35
      compute(i, data[j], ...);
36
      MPI_Rput(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
37
                 &put_req[j]);
38
     }
39
40
     MPI_Waitall(M, put_req, MPI_STATUSES_IGNORE);
41
     MPI_Win_unlock_all(win);
42
43
44
     Example 12.23 The following example constructs a distributed shared linked list using
45
     dynamic windows. Initially process 0 creates the head of the list, attaches it to the window,
46
     and broadcasts the pointer to all processes. All processes then concurrently append N new
47
     elements to the list. When a process attempts to attach its element to the tail of the
```

CHAPTER 12. ONE-SIDED COMMUNICATIONS

576

48

list it may discover that its tail pointer is stale and it must chase ahead to the new tail

before the element can be attached. This example requires some modification to work in an environment where the layout of the structures is different on different processes.

```
4
#define NUM_ELEMS 10
                                                                                     5
                                                                                     6
#define LLIST_ELEM_NEXT_RANK ( offsetof(llist_elem_t, next) + \
                                                                                     7
                                 offsetof(llist_ptr_t, rank) )
#define LLIST_ELEM_NEXT_DISP ( offsetof(llist_elem_t, next) + \
                                                                                     9
                                 offsetof(llist_ptr_t, disp) )
                                                                                     10
                                                                                     11
/* Linked list pointer */
                                                                                     12
typedef struct {
                                                                                     13
 MPI_Aint disp;
                                                                                     14
  int
           rank;
                                                                                     15
} llist_ptr_t;
                                                                                     16
                                                                                     17
/* Linked list element */
                                                                                     18
typedef struct {
                                                                                     19
 llist_ptr_t next;
                                                                                     20
 int value;
                                                                                    21
} llist_elem_t;
                                                                                    22
                                                                                    23
const llist_ptr_t nil = { (MPI_Aint) MPI_BOTTOM, -1 };
                                                                                     ^{24}
                                                                                     25
/* List of locally allocated list elements. */
                                                                                     26
static llist_elem_t **my_elems = NULL;
                                                                                     27
static int my_elems_size = 0;
                                                                                     28
static int my_elems_count = 0;
                                                                                     29
                                                                                     30
/* Allocate a new shared linked list element */
                                                                                     31
MPI_Aint alloc_elem(int value, MPI_Win win) {
                                                                                     32
 MPI_Aint disp;
                                                                                     33
  llist_elem_t *elem_ptr;
                                                                                    34
                                                                                    35
  /* Allocate the new element and register it with the window */
                                                                                    36
 MPI_Alloc_mem(sizeof(llist_elem_t), MPI_INFO_NULL, &elem_ptr);
                                                                                    37
  elem_ptr->value = value;
                                                                                     38
  elem_ptr->next = nil;
                                                                                     39
 MPI_Win_attach(win, elem_ptr, sizeof(llist_elem_t));
                                                                                     40
                                                                                     41
  /* Add the element to the list of local elements so we can free
                                                                                    42
     it later. */
                                                                                     43
  if (my_elems_size == my_elems_count) {
                                                                                     44
    my_elems_size += 100;
                                                                                     45
    my_elems = realloc(my_elems, my_elems_size*sizeof(void*));
                                                                                     46
  }
                                                                                     47
 my_elems[my_elems_count] = elem_ptr;
                                                                                     48
```

1

 $\mathbf{2}$

```
1
       my_elems_count++;
2
3
       MPI_Get_address(elem_ptr, &disp);
4
       return disp;
5
     }
6
7
     int main(int argc, char *argv[]) {
8
       int
                      procid, nproc, i;
9
       MPI_Win
                      llist_win;
10
                     head_ptr, tail_ptr;
       llist_ptr_t
11
12
       MPI_Init(&argc, &argv);
13
14
       MPI_Comm_rank(MPI_COMM_WORLD, &procid);
15
       MPI_Comm_size(MPI_COMM_WORLD, &nproc);
16
17
       MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &llist_win);
18
19
       /* Process 0 creates the head node */
20
       if (procid == 0)
21
         head_ptr.disp = alloc_elem(-1, llist_win);
22
       /* Broadcast the head pointer to everyone */
23
24
       head_ptr.rank = 0;
25
       MPI_Bcast(&head_ptr.disp, 1, MPI_AINT, 0, MPI_COMM_WORLD);
26
       tail_ptr = head_ptr;
27
28
       /* Lock the window for shared access to all targets */
29
       MPI_Win_lock_all(0, llist_win);
30
31
       /* All processes concurrently append NUM_ELEMS elements to the list */
32
       for (i = 0; i < NUM_ELEMS; i++) {</pre>
33
         llist_ptr_t new_elem_ptr;
34
         int success;
35
36
         /* Create a new list element and attach it to the window */
37
         new_elem_ptr.rank = procid;
38
         new_elem_ptr.disp = alloc_elem(procid, llist_win);
39
40
         /* Append the new node to the list. This might take multiple
41
            attempts if others have already appended and our tail pointer
42
            is stale. */
43
         do {
44
           llist_ptr_t next_tail_ptr = nil;
45
46
           MPI_Compare_and_swap((void*) &new_elem_ptr.rank, (void*) &nil.rank,
47
                (void*)&next_tail_ptr.rank, MPI_INT, tail_ptr.rank,
48
               MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_RANK),
```

. . .

```
1
        llist_win);
                                                                                    \mathbf{2}
                                                                                    3
    MPI_Win_flush(tail_ptr.rank, llist_win);
    success = (next_tail_ptr.rank == nil.rank);
                                                                                    4
                                                                                    5
                                                                                    6
    if (success) {
      MPI_Accumulate(&new_elem_ptr.disp, 1, MPI_AINT, tail_ptr.rank,
          MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP), 1,
          MPI_AINT, MPI_REPLACE, llist_win);
                                                                                    9
                                                                                   10
                                                                                   11
      MPI_Win_flush(tail_ptr.rank, llist_win);
      tail_ptr = new_elem_ptr;
                                                                                   12
                                                                                   13
    } else {
                                                                                   14
                                                                                   15
      /* Tail pointer is stale, fetch the displacement.
                                                            May take
                                                                                   16
         multiple tries if it is being updated. */
                                                                                   17
      do {
                                                                                   18
        MPI_Get_accumulate(NULL, 0, MPI_AINT, &next_tail_ptr.disp,
                                                                                   19
             1, MPI_AINT, tail_ptr.rank,
             MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP),
                                                                                   20
                                                                                   21
             1, MPI_AINT, MPI_NO_OP, llist_win);
                                                                                   22
        MPI_Win_flush(tail_ptr.rank, llist_win);
                                                                                   23
                                                                                   ^{24}
      } while (next_tail_ptr.disp == nil.disp);
      tail_ptr = next_tail_ptr;
                                                                                   25
                                                                                   26
    }
  } while (!success);
                                                                                   27
}
                                                                                   28
                                                                                   29
                                                                                   30
MPI_Win_unlock_all(llist_win);
MPI_Barrier(MPI_COMM_WORLD);
                                                                                   31
                                                                                   32
                                                                                   33
/* Free all the elements in the list */
for ( ; my_elems_count > 0; my_elems_count--) {
                                                                                   34
  MPI_Win_detach(llist_win,my_elems[my_elems_count-1]);
                                                                                   35
  MPI_Free_mem(my_elems[my_elems_count-1]);
                                                                                   36
                                                                                   37
}
                                                                                   38
MPI_Win_free(&llist_win);
                                                                                   39
                                                                                   40
                                                                                   41
                                                                                   42
                                                                                   43
                                                                                   44
                                                                                   45
                                                                                   46
```



Chapter 13

External Interfaces

13.1 Introduction

This chapter contains calls used to create **generalized requests**, which allow users to create new nonblocking operations with an interface similar to what is present in MPI. These calls can be used to layer new functionality on top of MPI. Next, Section 13.3 deals with setting the information found in **status**. This functionality is needed for generalized requests.

13.2 Generalized Requests

The goal of generalized requests is to allow users to define new nonblocking operations. Such an outstanding nonblocking operation is represented by a (generalized) request. A fundamental property of nonblocking operations is that progress toward the completion of this operation occurs asynchronously, i.e., concurrently with normal program execution. Typically, this requires execution of code concurrently with the execution of the user code, e.g., in a separate thread or in a signal handler. Operating systems provide a variety of mechanisms in support of concurrent execution. MPI does not attempt to standardize or to replace these mechanisms: it is assumed programmers who wish to define new asynchronous operations will use the mechanisms provided by the underlying operating system. Thus, the calls in this section only provide a means for defining the effect of MPI calls such as MPI_WAIT or MPI_CANCEL when they apply to generalized requests, and for signaling to MPI the completion of a generalized operation.

Rationale. It is tempting to also define an MPI standard mechanism for achieving concurrent execution of user-defined nonblocking operations. However, it is difficult to define such a mechanism without consideration of the specific mechanisms used in the operating system. The Forum feels that concurrency mechanisms are a proper part of the underlying operating system and should not be standardized by MPI; the MPI standard should only deal with the interaction of such mechanisms with MPI. (*End of rationale.*)

For a regular request, the operation associated with the request is performed by the MPI implementation, and the operation completes without intervention by the application. For a generalized request, the operation associated with the request is performed by the application; therefore, the application must notify MPI through a call to

 $45 \\ 46$

```
1
     MPI_GREQUEST_COMPLETE when the operation completes. MPI maintains the "comple-
\mathbf{2}
     tion" status of generalized requests. Any other request state has to be maintained by the
3
     user.
4
          A new generalized request is started with
5
6
     MPI_GREQUEST_START(query_fn, free_fn, cancel_fn, extra_state, request)
7
8
       IN
                                             callback function invoked when request status is
                 query_fn
9
                                             queried (function)
10
       IN
                 free_fn
                                             callback function invoked when request is freed
11
                                             (function)
12
                                             callback function invoked when request is cancelled
       IN
                 cancel_fn
13
                                             (function)
14
15
       IN
                 extra_state
                                             extra state
16
       OUT
                                             generalized request (handle)
                 request
17
18
     C binding
19
     int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
20
                    MPI_Grequest_free_function *free_fn,
21
                    MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
22
                    MPI_Request *request)
23
^{24}
     Fortran 2008 binding
25
     MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
26
                     ierror)
27
          PROCEDURE(MPI_Grequest_query_function), INTENT(IN) :: query_fn
28
          PROCEDURE(MPI_Grequest_free_function), INTENT(IN) :: free_fn
29
          PROCEDURE(MPI_Grequest_cancel_function), INTENT(IN) :: cancel_fn
30
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
31
          TYPE(MPI_Request), INTENT(OUT) :: request
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
     Fortran binding
34
     MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
35
                     IERROR)
36
          EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
37
          INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
38
          INTEGER REQUEST, IERROR
39
40
41
           Advice to users.
                              Note that a generalized request is of the same type as regular
42
           requests, in C and Fortran. (End of advice to users.)
43
         The call starts a generalized request and returns a handle to it in request.
44
          The syntax and meaning of the callback functions are listed below. All callback func-
45
     tions are passed the extra_state argument that was associated with the request by the
46
47
     starting call MPI_GREQUEST_START; extra_state can be used to maintain user-defined
48
     state for the request.
```

```
1
    In C, the query function is
                                                                                          2
typedef int MPI_Grequest_query_function(void *extra_state,
               MPI_Status *status);
in Fortran with the mpi_f08 module
ABSTRACT INTERFACE
  SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
    TYPE(MPI_Status) :: status
                                                                                          9
    INTEGER :: ierror
                                                                                          10
                                                                                          11
in Fortran with the mpi module and mpif.h
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
                                                                                          12
                                                                                          13
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                          14
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
                                                                                          15
    The query_fn function computes the status that should be returned for the generalized
                                                                                          16
request. The status also includes information about successful/unsuccessful cancellation of
                                                                                          17
the request (result to be returned by MPI_TEST_CANCELLED).
                                                                                          18
    The query_fn callback is invoked by the MPI_{WAIT|TEST}{ANY|SOME|ALL} call that
                                                                                          19
completed the generalized request associated with this callback. The callback function is
                                                                                          20
also invoked by calls to MPI_REQUEST_GET_STATUS, if the request is complete when
                                                                                          21
the call occurs. In both cases, the callback is passed a reference to the corresponding
                                                                                          22
status variable passed by the user to the MPI call; the status set by the callback function
                                                                                          23
is returned by the MPI call. If the user provided MPI_STATUS_IGNORE or
                                                                                          ^{24}
MPI_STATUSES_IGNORE to the MPI function that causes query_fn to be called, then MPI
                                                                                          25
will pass a valid status object to query fn, and this status will be ignored upon return of the
                                                                                          26
callback function. Note that query_fn is invoked only after MPI_GREQUEST_COMPLETE
                                                                                          27
is called on the request; it may be invoked several times for the same generalized request,
                                                                                          28
e.g., if the user calls MPI_REQUEST_GET_STATUS several times for this request. Note also
                                                                                          29
that a call to MPI_{WAIT|TEST}{SOME|ALL} may cause multiple invocations of query_fn
                                                                                          30
callback functions, one for each generalized request that is completed by the MPI call. The
                                                                                          ^{31}
order of these invocations is not specified by MPI.
                                                                                          32
    In C, the free function is
                                                                                          33
typedef int MPI_Grequest_free_function(void *extra_state);
                                                                                          34
                                                                                          35
in Fortran with the mpi_f08 module
                                                                                          36
ABSTRACT INTERFACE
                                                                                          37
  SUBROUTINE MPI_Grequest_free_function(extra_state, ierror)
                                                                                          38
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
                                                                                          39
    INTEGER :: ierror
                                                                                          40
in Fortran with the mpi module and mpif.h
                                                                                          41
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
                                                                                          42
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                          43
    INTEGER IERROR
                                                                                          44
                                                                                          45
The free_fn function is invoked to clean up user-allocated resources when the generalized
                                                                                          46
request is freed.
                                                                                          47
```

1 The free_fn callback is invoked by the MPI_{WAIT|TEST}{ANY|SOME|ALL} call that $\mathbf{2}$ completed the generalized request associated with this callback. free_fn is invoked after 3 the call to query_fn for the same request. However, if the MPI call completed multiple 4 generalized requests, the order in which free_fn callback functions are invoked is not specified $\mathbf{5}$ by MPI.

6 The free_fn callback is also invoked for generalized requests that are freed by a call 7to MPI_REQUEST_FREE (no call to MPI_{WAIT|TEST}{ANY|SOME|ALL} will occur for 8 such a request). In this case, the callback function will be called either in the MPI call 9 MPI_REQUEST_FREE(request), or in the MPI call MPI_GREQUEST_COMPLETE(request), 10 whichever happens last, i.e., in this case the actual freeing code is executed as soon as both 11calls MPI_REQUEST_FREE and MPI_GREQUEST_COMPLETE have occurred. The request 12is not deallocated until after free_fn completes. Note that free_fn will be invoked only once 13per request by a correct program.

Advice to users. Calling MPI_REQUEST_FREE(request) will cause the request handle 15to be set to MPI_REQUEST_NULL. This handle to the generalized request is no longer 16valid. However, user copies of this handle are valid until after free_fn completes since MPI does not deallocate the object until then. Since free_fn is not called until after MPI_GREQUEST_COMPLETE, the user copy of the handle can be used to make this 19 call. Users should note that MPI will deallocate the object after free_fn executes. At 20this point, user copies of the request handle no longer point to a valid request. MPI will not set user copies to MPI_REQUEST_NULL in this case, so it is up to the user to avoid accessing this stale handle. This is a special case in which MPI defers deallocating the 23object until a later time that is known by the user. (End of advice to users.) 24

In C, the cancel function is

typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);

```
28
     in Fortran with the mpi_f08 module
```

29ABSTRACT INTERFACE

30 SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror) 31INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state 32 LOGICAL :: complete 33

INTEGER :: ierror 34

```
in Fortran with the mpi module and mpif.h
35
```

```
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
36
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
37
         LOGICAL COMPLETE
38
```

INTEGER IERROR 39

40 The cancel_fn function is invoked to start the cancelation of a generalized request. It 41 is called by MPI_CANCEL(request). MPI passes complete = true to the callback function 42if MPI_GREQUEST_COMPLETE was already called on the request, and complete = false 43otherwise.

44All callback functions return an error code. The code is passed back and dealt with as 45appropriate for the error code by the MPI function that invoked the callback function. For 46example, if error codes are returned then the error code returned by the callback function 47will be returned by the MPI function that invoked the callback function. In the case of 48an MPI_{WAIT|TEST}{ANY} call that invokes both query_fn and free_fn, the MPI call will

Unofficial Draft for Comment Only

14

17

18

21

22

25

26

return the error code returned by the last callback, namely free_fn. If one or more of the requests in a call to MPI_{WAIT|TEST}{SOME|ALL} failed, then the MPI call will return MPI_ERR_IN_STATUS. In such a case, if the MPI call was passed an array of statuses, then MPI will return in each of the statuses that correspond to a completed generalized request the error code returned by the corresponding invocation of its free_fn callback function. However, if the MPI function was passed MPI_STATUSES_IGNORE, then the individual error codes returned by each callback functions will be lost.

Advice to users. query_fn must not set the error field of status since query_fn may be called by MPI_WAIT or MPI_TEST, in which case the error field of status should not change. The MPI library knows the "context" in which query_fn is invoked and can decide correctly when to put the returned error code in the error field of status. (End of advice to users.)

MPI_GREQUEST_COMPLETE(request)

INOUT request

generalized request (handle)

C binding

int MPI_Grequest_complete(MPI_Request request)

Fortran 2008 binding

```
MPI_Grequest_complete(request, ierror)
    TYPE(MPI_Request), INTENT(IN) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

MPI_GREQUEST_COMPLETE(REQUEST, IERROR)

INTEGER REQUEST, IERROR

The call informs MPI that the operations represented by the generalized request request are complete (see definitions in Section 2.4). A call to MPI_WAIT(request, status) will return and a call to MPI_TEST(request, flag, status) will return flag = true only after a call to MPI_GREQUEST_COMPLETE has declared that these operations are complete.

MPI imposes no restrictions on the code executed by the callback functions. However, new nonblocking operations should be defined so that the general semantic rules about MPI calls such as MPI_TEST, MPI_REQUEST_FREE, or MPI_CANCEL still hold. For example, these calls are supposed to be local and nonblocking. Therefore, the callback functions query_fn, free_fn, or cancel_fn should invoke blocking MPI communication calls only if the context is such that these calls are guaranteed to return in finite time. Once MPI_CANCEL is invoked, the cancelled operation should complete in finite time, irrespective of the state of other processes (the operation has acquired "local" semantics). It should either succeed, or fail without side-effects. The user should guarantee these same properties for newly defined operations.

Advice to implementors. A call to MPI_GREQUEST_COMPLETE may unblock a blocked user process/thread. The MPI library should ensure that the blocked user computation will resume. (*End of advice to implementors.*)

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\frac{44}{45}$

```
13.2.1 Examples
```

Example 13.1 This example shows the code for a user-defined reduce operation on an int using a binary tree: each non-root node receives two messages, sums them, and sends them up. We assume that no status is returned and that the operation cannot be cancelled.

```
7
     typedef struct {
8
        MPI_Comm comm;
9
        int tag;
10
        int root;
11
        int valin:
12
        int *valout;
13
        MPI_Request request;
14
        } ARGS;
15
16
17
     int myreduce(MPI_Comm comm, int tag, int root,
18
                   int valin, int *valout, MPI_Request *request)
19
     {
20
        ARGS *args;
21
        pthread_t thread;
22
23
        /* start request */
24
        MPI_Grequest_start(query_fn, free_fn, cancel_fn, NULL, request);
25
26
        args = (ARGS*)malloc(sizeof(ARGS));
27
         args->comm = comm;
28
        args \rightarrow tag = tag;
29
        args->root = root;
30
        args->valin = valin;
31
         args->valout = valout;
32
         args->request = *request;
33
34
        /* spawn thread to handle request */
35
         /* The availability of the pthread_create call is system dependent */
36
        pthread_create(&thread, NULL, reduce_thread, args);
37
38
        return MPI_SUCCESS;
39
     }
40
41
     /* thread code */
42
     void* reduce_thread(void *ptr)
43
     ſ
44
        int lchild, rchild, parent, lval, rval, val;
45
        MPI_Request req[2];
46
        ARGS *args;
47
48
        args = (ARGS*)ptr;
```

1

2 3

4

5

```
2
   /* compute left and right child and parent in tree; set
      to MPI_PROC_NULL if does not exist */
   /* code not shown */
   . . .
  MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
  MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
  MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
                                                                                    a
                                                                                   10
  val = lval + args->valin + rval;
                                                                                   11
  MPI_Send(&val, 1, MPI_INT, parent, args->tag, args->comm);
   if (parent == MPI_PROC_NULL) *(args->valout) = val;
                                                                                   12
  MPI_Grequest_complete((args->request));
                                                                                   13
                                                                                   14
  free(ptr);
                                                                                   15
  return(NULL);
                                                                                   16
}
                                                                                   17
                                                                                   18
int query_fn(void *extra_state, MPI_Status *status)
                                                                                   19
Ł
   /* always send just one int */
                                                                                   20
                                                                                   21
  MPI_Status_set_elements(status, MPI_INT, 1);
   /* can never cancel so always true */
                                                                                   22
                                                                                   23
  MPI_Status_set_cancelled(status, 0);
                                                                                   24
  /* choose not to return a value for this */
                                                                                   25
   status->MPI_SOURCE = MPI_UNDEFINED;
                                                                                   26
   /* tag has no meaning for this generalized request */
   status->MPI_TAG = MPI_UNDEFINED;
                                                                                   27
  /* this generalized request never fails */
                                                                                   28
                                                                                   29
  return MPI_SUCCESS;
}
                                                                                   30
                                                                                   31
                                                                                   32
                                                                                   33
int free_fn(void *extra_state)
                                                                                   34
ł
   /* this generalized request does not need to do any freeing */
                                                                                   35
   /* as a result it never fails here */
                                                                                   36
                                                                                   37
  return MPI_SUCCESS;
}
                                                                                   38
                                                                                   39
                                                                                   40
                                                                                   41
int cancel_fn(void *extra_state, int complete)
                                                                                   42
{
   /* This generalized request does not support cancelling.
                                                                                   43
                                                                                   44
      Abort if not already done. If done then treat as if cancel failed.*/
                                                                                   45
   if (!complete) {
                                                                                   46
     fprintf(stderr,
                                                                                   47
             "Cannot cancel generalized request - aborting program\n");
                                                                                   48
     MPI_Abort(MPI_COMM_WORLD, 99);
```

```
}
return MPI_SUCCESS;
}
```

13.3 Associating Information with Status

MPI supports several different types of requests besides those for point-to-point operations. These range from MPI calls for I/O to generalized requests. It is desirable to allow these calls to use the same request mechanism, which allows one to wait or test on different types of requests. However, MPI_{TEST|WAIT}{ANY|SOME|ALL} returns a status with information about the request. With the generalization of requests, one needs to define what information will be returned in the status object.

Each MPI call fills in the appropriate fields in the status object. Any unused fields will have undefined values. A call to MPI_{TEST|WAIT}{ANY|SOME|ALL} can modify any of the fields in the status object. Specifically, it can modify fields that are undefined. The fields with meaningful values for a given request are defined in the sections with the new request.

Generalized requests raise additional considerations. Here, the user provides the functions to deal with the request. Unlike other MPI calls, the user needs to provide the information to be returned in the status. The status argument is provided directly to the callback function where the status needs to be set. Users can directly set the values in 3 of the 5 status values. The count and cancel fields are opaque. To overcome this, these calls are provided:

25 26

MPI_STATUS_SET_ELEMENTS(status, datatype, count)

			. ,	
27 28	INOUT	status	status with which to associate count (Status)	
29	IN	datatype	datatype associated with count (handle)	
30	IN	count	number of elements to associate with status (integer)	
31				
32	C binding	g		
33	int MPI_S	- Status_set_elements(MI	PI_Status *status, MPI_Datatype datatype,	
34 35		int count)		
35 36	Fortran 2	008 binding		
37			s, datatype, count, ierror)	
38	TYPE(MPI_Status), INTENT(INOUT) :: status			
39	TYPE(MPI_Datatype), INTENT(IN) :: datatype			
40	INTEGER, INTENT(IN) :: count			
41	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
42	Fortran k	binding		
43	MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)			
44			_SIZE), DATATYPE, COUNT, IERROR	
45				
46 47				
47				

1

 $\mathbf{2}$

3

4 5

> 6 7

8

9

10

11

MPI_STAT	US_SET_ELEMENTS_X(stat	us, datatype, count)	1
INOUT	status	status with which to associate count (Status)	2
IN	datatype	datatype associated with count (handle)	3 4
IN	count	number of elements to associate with status (integer)	5
	count	number of elements to appendie with status (integer)	6
C bindin	or and the second se		7
	-	_Status *status, MPI_Datatype datatype,	8
	MPI_Count count)		9
Fortran 2	2008 binding		10 11
	0	datatype, count, ierror)	12
TYPE	(MPI_Status), INTENT(INOU	T) :: status	13
	(MPI_Datatype), INTENT(IN		14
	GER(KIND=MPI_COUNT_KIND),		15
INTEC	ER, OPTIONAL, INTENT(OUT) :: lerror	16 17
Fortran b	0		18
		DATATYPE, COUNT, IERROR)	19
	GER STATUS(MPI_STATUS_SIZ GER(KIND=MPI_COUNT_KIND)		20
			21
	· · · ·	e part of status so that a call to	22
	_ELEMENTS or MPI_GET_EL a compatible value.	EMENTS_X will return count. MPI_GET_COUNT	23 24
will febuili	a compatible value.		24 25
Ratio	onale. The number of eleme	ents is set instead of the count because the former	26
can o	deal with a nonintegral numb	er of datatypes. (End of rationale.)	27
A and	accurate call to MDL CET C	OUNT(status, datature, sount)	28
MPL CET ELEMENTS(status datature count) or			29
MPL CET ELEMENTS X(status, datations, count) must use a datations argument that has			30 31
		e argument that was used in the call to	32
MPI_STAT	MPI_STATUS_SET_ELEMENTS or MPI_STATUS_SET_ELEMENTS_X.		
Dati	Rationale. The requirement of matching type signatures for these calls is similar		
	-	matching type signatures for these calls is similar a count is set by a receive operation: in that case,	35
		PI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X	36
		e signature as the datatype used in the receive call.	37 38
(End	l of rationale.)		39
			40
			41
MPI_STAT	US_SET_CANCELLED(status	, flag)	42
INOUT	status	status with which to associate cancel flag (Status)	43
IN	flag	if true, indicates request was cancelled (logical)	44 45
	<u> </u>	, 1	46
C binding	g		47
int MPI_S	Status_set_cancelled(MPI_	Status *status, int flag)	48

Unofficial Draft for Comment Only

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Status_set_cancelled(status, flag, ierror)
3
          TYPE(MPI_Status), INTENT(INOUT) :: status
4
          LOGICAL, INTENT(IN) :: flag
5
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     Fortran binding
7
     MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
8
          INTEGER STATUS(MPI_STATUS_SIZE), IERROR
9
          LOGICAL FLAG
10
11
          If flag is set to true then a subsequent call to MPI_TEST_CANCELLED(status, flag) will
12
     also return flag = true, otherwise it will return false.
13
14
           Advice to users.
                              Users are advised not to reuse the status fields for values other
15
           than those for which they were intended. Doing so may lead to unexpected results
16
           when using the status object. For example, calling MPI_GET_ELEMENTS may cause
17
           an error if the value is out of range or it may be impossible to detect such an error.
18
           The extra_state argument provided with a generalized request can be used to return
19
           information that does not logically belong in status. Furthermore, modifying the
           values in a status set internally by MPI, e.g., MPI_RECV, may lead to unpredictable
20
           results and is strongly discouraged. (End of advice to users.)
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

Chapter 14

I/O

14.1 Introduction

POSIX provides a model of a widely portable file system, but the portability and optimization needed for parallel I/O cannot be achieved with the POSIX interface.

The significant optimizations required for efficiency (e.g., grouping [54], collective buffering [8, 16, 55, 59, 66], and disk-directed I/O [48]) can only be implemented if the parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. In addition, further efficiencies can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on storage devices (disks). The I/O environment described in this chapter provides these facilities.

Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach in which data partitioning is expressed using derived datatypes. Compared to a limited set of predefined access patterns, this approach has the advantage of added flexibility and expressiveness.

14.1.1 Definitions

- file An MPI file is an ordered collection of typed data items. MPI supports random or sequential access to any integral set of these items. A file is opened collectively by a group of processes. All collective I/O calls on a file are collective over this group.
- **displacement** A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a *view* begins. Note that a "file displacement" is distinct from a "typemap displacement."
- etype An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes. Depending on context, the term "etype" is used to describe one of three aspects of an elementary datatype: a particular MPI type, a data item of that type, or the extent of that type.

 24

filetype A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. The displacements in the typemap of the filetype are not required to be distinct, but they must be non-negative and monotonically nondecreasing.

view A *view* defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The pattern of repetition is defined to be the same pattern that MPI_TYPE_CONTIGUOUS would produce if it were passed the filetype and an arbitrarily large count. Figure 14.1 shows how the tiling works; note that the filetype in this example must have explicit lower and upper bounds set in order for the initial and final holes to be repeated in the view. Views can be changed by the user during program execution. The default view is a linear byte stream (displacement is zero, etype and filetype equal to MPI_BYTE).

18	etype
19	
20	filetype
21	holes —
22	tiling a file with the filetype:
23	uning a me with the metype.
24	
25	displacement accessible data
26	
27	Figure 14.1: Etypes and filetypes
28	A meun of processes can use complementary views to achieve a global data distribution
29	A group of processes can use complementary views to achieve a global data distribution
30	such as a scatter/gather pattern (see Figure 14.2).
31	etype
32	process 0 filetype
33	
34	process 1 filetype
35	process 2 filetype
36	tiling a file with the filetomest
37	tiling a file with the filetypes:
38	
39	displacement
40	
41	Figure 14.2: Partitioning a file among parallel processes
42	

offset An offset is a position in the file relative to the current view, expressed as a count of etypes. Holes in the view's filetype are skipped when calculating this position. Offset 0 is the location of the first etype visible in the view (after skipping the displacement and any initial holes in the view). For example, an offset of 2 for process 1 in Figure 14.2 is the position of the eighth etype in the file after the displacement. An "explicit offset" is an offset that is used as an argument in explicit data access routines.

Unofficial Draft for Comment Only

- file size and end of file The *size* of an MPI file is measured in bytes from the beginning of the file. A newly created file has a size of zero bytes. Using the size as an absolute displacement gives the position of the byte immediately following the last byte in the file. For any given view, the end of file is the offset of the first etype accessible in the current view starting after the last byte in the file.
- file pointer A file pointer is an implicit offset maintained by MPI. "Individual file pointers" are file pointers that are local to each process that opened the file. A "shared file pointer" is a file pointer that is shared by the group of processes that opened the file.
- file handle A file handle is an opaque object created by MPI_FILE_OPEN and freed by MPI_FILE_CLOSE. All operations on an open file reference the file through the file handle.

14.2 File Manipulation

14.2.1 Opening a File

			15
MPI_FILE	_OPEN(comm, filename, am	ode, info, fh)	20
IN	comm	communicator (handle)	21
IN	filename	name of file to open (string)	22 23
IN	amode	file access mode (integer)	23
IN	info	info object (handle)	25
	IIIIO	mo object (nancie)	26
OUT	fh	new file handle (handle)	27
			28
C bindin	lg		29
<pre>int MPI_File_open(MPI_Comm comm, const char *filename, int amode,</pre>		30	
	MPI_Info info, MPI	_File *fh)	31
Fortran	2008 binding		32
MPI_File_open(comm, filename, amode, info, fh, ierror)			33
TYPE(MPI_Comm), INTENT(IN) :: comm			34
	CHARACTER(LEN=*), INTENT(IN) :: filename		
	INTEGER, INTENT(IN) :: amode		
	TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_File), INTENT(OUT) :: fh		38
	GER, OPTIONAL, INTENT(OUT)		39
T 1/ T C/	GER, OF ITOWAL, INTENI (U	01/ I <u>CII</u> OI	40

Fortran binding

MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR) INTEGER COMM, AMODE, INFO, FH, IERROR CHARACTER*(*) FILENAME

MPI_FILE_OPEN opens the file identified by the file name filename on all processes in the comm communicator group. MPI_FILE_OPEN is a collective routine: all processes must provide the same value for amode, and all processes must provide filenames that reference

 $\mathbf{2}$

 $\overline{7}$

1 the same file. (Values for info may vary.) comm must be an intracommunicator; it is $\mathbf{2}$ erroneous to pass an intercommunicator to MPI_FILE_OPEN. Errors in MPI_FILE_OPEN 3 are raised using the default file error handler (see Section 14.7). When using the World 4 Model (Section 11.1), a process can open a file independently of other processes by using 5the MPI_COMM_SELF communicator. Applications using the Sessions Model (Section 11.3) 6 can achieve the same result using communicators created from the "mpi://SELF" process $\overline{7}$ set. The file handle returned, fh, can be subsequently used to access the file until the file is 8 closed using MPI_FILE_CLOSE. Before calling MPI_FINALIZE, the user is required to close 9 (via MPI_FILE_CLOSE) all files that were opened with MPI_FILE_OPEN. Note that the 10 communicator comm is unaffected by MPI_FILE_OPEN and continues to be usable in all 11MPI routines (e.g., MPI_SEND). Furthermore, the use of comm will not interfere with I/O 12behavior.

The format for specifying the file name in the filename argument is implementation dependent and must be documented by the implementation.

Advice to implementors. An implementation may require that filename include a string or strings specifying additional information about the file. Examples include the type of filesystem (e.g., a prefix of ufs:), a remote hostname (e.g., a prefix of machine.univ.edu:), or a file password (e.g., a suffix of /PASSWORD=SECRET). (End of advice to implementors.)

Advice to users. On some implementations of MPI, the file namespace may not be identical from all processes of all applications. For example, "/tmp/foo" may denote different files on different processes, or a single file may have many names, dependent on process location. The user is responsible for ensuring that a single file is referenced by the filename argument, as it may be impossible for an implementation to detect this type of namespace error. (*End of advice to users.*)

27 28 29

30

 31

32

33

34

35 36

37 38

39

40 41

42 43

44

 $45 \\ 46$

47 48

15

16

17

18

19

20 21

22

23

24

25

26

Initially, all processes view the file as a linear byte stream, and each process views data in its own native representation (no data representation conversion is performed). (POSIX files are linear byte streams in the native representation.) The file view can be changed via the MPI_FILE_SET_VIEW routine.

The following access modes are supported (specified in amode, a bit vector OR of the following integer constants):

- MPI_MODE_RDONLY read only,
- MPI_MODE_RDWR reading and writing,
- MPI_MODE_WRONLY write only,
- MPI_MODE_CREATE create the file if it does not exist,
- MPI_MODE_EXCL error if creating file that already exists,
- MPI_MODE_DELETE_ON_CLOSE delete file on close,
- MPI_MODE_UNIQUE_OPEN file will not be concurrently opened elsewhere,
- MPI_MODE_SEQUENTIAL file will only be accessed sequentially,
 - MPI_MODE_APPEND set initial position of all file pointers to end of file.

Advice to users. C users can use bit vector OR (|) to combine these constants; Fortran 90 users can use the bit vector IOR intrinsic. Fortran 77 users can use (nonportably) bit vector IOR on systems that support it. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition.). (End of advice to users.)

Advice to implementors. The values of these constants must be defined such that the bitwise OR and the sum of any distinct set of these constants is equivalent. (*End of advice to implementors.*)

The modes MPI_MODE_RDONLY, MPI_MODE_RDWR, MPI_MODE_WRONLY, MPI_MODE_CREATE, and MPI_MODE_EXCL have identical semantics to their POSIX counterparts [44]. Exactly one of MPI_MODE_RDONLY, MPI_MODE_RDWR, or MPI_MODE_WRONLY, must be specified. It is erroneous to specify MPI_MODE_CREATE or MPI_MODE_EXCL in conjunction with MPI_MODE_RDONLY; it is erroneous to specify MPI_MODE_SEQUENTIAL together with MPI_MODE_RDWR.

The MPI_MODE_DELETE_ON_CLOSE mode causes the file to be deleted (equivalent to performing an MPI_FILE_DELETE) when the file is closed.

The MPI_MODE_UNIQUE_OPEN mode allows an implementation to optimize access by eliminating the overhead of file locking. It is erroneous to open a file in this mode unless the file will not be concurrently opened elsewhere.

Advice to users. For MPI_MODE_UNIQUE_OPEN, not opened elsewhere includes both inside and outside the MPI environment. In particular, one needs to be aware of potential external events which may open files (e.g., automated backup facilities). When MPI_MODE_UNIQUE_OPEN is specified, the user is responsible for ensuring that no such external events take place. (End of advice to users.)

The MPI_MODE_SEQUENTIAL mode allows an implementation to optimize access to some sequential devices (tapes and network streams). It is erroneous to attempt nonsequential access to a file that has been opened in this mode.

Specifying MPI_MODE_APPEND only guarantees that all shared and individual file pointers are positioned at the initial end of file when MPI_FILE_OPEN returns. Subsequent positioning of file pointers is application dependent. In particular, the implementation does not ensure that all writes are appended.

Errors related to the access mode are raised in the class MPI_ERR_AMODE.

The info argument is used to provide information regarding file access patterns and file system specifics (see Section 14.2.8). The constant MPI_INFO_NULL can be used when no info needs to be specified.

Advice to users. Some file attributes are inherently implementation dependent (e.g., file permissions). These attributes must be set using either the info argument or facilities outside the scope of MPI. (*End of advice to users.*)

Files are opened by default using nonatomic mode file consistency semantics (see Section 14.6.1). The more stringent atomic mode consistency semantics, required for atomicity of conflicting accesses, can be set using MPI_FILE_SET_ATOMICITY.

Unofficial Draft for Comment Only

 $\overline{7}$

 24

```
1
     14.2.2 Closing a File
\mathbf{2}
3
4
     MPI_FILE_CLOSE(fh)
5
       INOUT
                 fh
                                              file handle (handle)
6
7
     C binding
8
     int MPI_File_close(MPI_File *fh)
9
10
     Fortran 2008 binding
11
     MPI_File_close(fh, ierror)
12
          TYPE(MPI_File), INTENT(INOUT) :: fh
13
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     Fortran binding
15
16
     MPI_FILE_CLOSE(FH, IERROR)
17
          INTEGER FH, IERROR
18
          MPI_FILE_CLOSE first synchronizes file state (equivalent to performing an
19
     MPI_FILE_SYNC), then closes the file associated with fh. The file is deleted if it was
20
     opened with access mode MPI_MODE_DELETE_ON_CLOSE (equivalent to performing an
21
     MPI_FILE_DELETE). MPI_FILE_CLOSE is a collective routine.
22
23
           Advice to users. If the file is deleted on close, and there are other processes currently
^{24}
           accessing the file, the status of the file and the behavior of future accesses by these
25
           processes are implementation dependent. (End of advice to users.)
26
27
          The user is responsible for ensuring that all outstanding nonblocking requests and
28
     split collective operations associated with fh made by a process have completed before that
29
     process calls MPI_FILE_CLOSE.
30
          The MPI_FILE_CLOSE routine deallocates the file handle object and sets fh to
^{31}
     MPI_FILE_NULL.
32
33
     14.2.3 Deleting a File
34
35
36
     MPI_FILE_DELETE(filename, info)
37
       IN
                 filename
                                              name of file to delete (string)
38
39
       IN
                 info
                                              info object (handle)
40
41
     C binding
42
     int MPI_File_delete(const char *filename, MPI_Info info)
43
     Fortran 2008 binding
44
     MPI_File_delete(filename, info, ierror)
45
46
          CHARACTER(LEN=*), INTENT(IN) :: filename
47
          TYPE(MPI_Info), INTENT(IN) :: info
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

Fortran binding

```
MPI_FILE_DELETE(FILENAME, INFO, IERROR)
CHARACTER*(*) FILENAME
INTEGER INFO, IERROR
```

MPI_FILE_DELETE deletes the file identified by the file name filename. If the file does not exist, MPI_FILE_DELETE raises an error in the class MPI_ERR_NO_SUCH_FILE.

The info argument can be used to provide information regarding file system specifics (see Section 14.2.8). The constant MPI_INFO_NULL refers to the null info, and can be used when no info needs to be specified.

If a process currently has the file open, the behavior of any access to the file (as well as the behavior of any outstanding accesses) is implementation dependent. In addition, whether an open file is deleted or not is also implementation dependent. If the file is not deleted, an error in the class MPI_ERR_FILE_IN_USE or MPI_ERR_ACCESS will be raised. Errors are raised using the default file error handler (see Section 14.7).

14.2.4 Resizing a File

MPI_FILE_	_SET_SIZE(fh, size)		
INOUT	fh	file handle (handle)	
IN	size	size to truncate or expand file (integer)	
C binding			
nt MPI_File_set_size(MPI_File fh, MPI_Offset size)			
	2008 binding set_size(fh, size	, ierror)	

```
TYPE(MPI_File), INTENT(IN) :: fh
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
INTEGER FH, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) SIZE
```

MPI_FILE_SET_SIZE resizes the file associated with the file handle fh. size is measured in bytes from the beginning of the file. MPI_FILE_SET_SIZE is collective; all processes in the group must pass identical values for size.

If size is smaller than the current file size, the file is truncated at the position defined by size. The implementation is free to deallocate file blocks located beyond this position.

If size is larger than the current file size, the file size becomes size. Regions of the file that have been previously written are unaffected. The values of data in the new regions in the file (those locations with displacements between old file size and size) are undefined. It is implementation dependent whether the MPI_FILE_SET_SIZE routine allocates file space — use MPI_FILE_PREALLOCATE to force file space to be reserved.

MPI_FILE_SET_SIZE does not affect the individual file pointers or the shared file

Unofficial Draft for Comment Only

 $\mathbf{2}$

3

 $\frac{4}{5}$

6

7

8

9

10

11

12

13

14

15 16

30

31

32

33

34

35

36 37

38

39

40

41

42

43

44

45

46

47

```
1
      pointer. If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is
\mathbf{2}
      erroneous to call this routine.
3
           Advice to users.
                              It is possible for the file pointers to point beyond the end of file
4
           after a MPI_FILE_SET_SIZE operation truncates a file. This is valid, and equivalent
5
           to seeking beyond the current end of file. (End of advice to users.)
6
7
          All nonblocking requests and split collective operations on fh must be completed before
8
      calling MPI_FILE_SET_SIZE. Otherwise, calling MPI_FILE_SET_SIZE is erroneous. As far
9
      as consistency semantics are concerned, MPI_FILE_SET_SIZE is a write operation that
10
      conflicts with operations that access bytes at displacements between the old and new file
11
      sizes (see Section 14.6.1).
12
13
      14.2.5 Preallocating Space for a File
14
15
                                                                                  16
17
     MPI_FILE_PREALLOCATE(fh, size)
18
                                               file handle (handle)
       INOUT
                  fh
19
       IN
                                               size to preallocate file (integer)
                 size
20
21
22
      C binding
23
      int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
^{24}
      Fortran 2008 binding
25
     MPI_File_preallocate(fh, size, ierror)
26
          TYPE(MPI_File), INTENT(IN) :: fh
27
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
28
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
      Fortran binding
^{31}
      MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
32
          INTEGER FH, IERROR
33
          INTEGER(KIND=MPI_OFFSET_KIND) SIZE
34
          MPI_FILE_PREALLOCATE ensures that storage space is allocated for the first size bytes
35
      of the file associated with fh. MPI_FILE_PREALLOCATE is collective; all processes in the
36
      group must pass identical values for size. Regions of the file that have previously been
37
      written are unaffected. For newly allocated regions of the file, MPI_FILE_PREALLOCATE
38
      has the same effect as writing undefined data. If size is larger than the current file size, the
39
      file size increases to size. If size is less than or equal to the current file size, the file size is
40
     unchanged.
41
```

The treatment of file pointers, pending nonblocking accesses, and file consistency is the same as with MPI_FILE_SET_SIZE. If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call this routine.

Advice to users. In some implementations, file preallocation may be time-consuming. (End of advice to users.)

47 48

45

14.2.6 Querying the Size of a File	1
	2
	3
MPI_FILE_GET_SIZE(fh, size)	4
IN fh file handle (handle)	5
	6
OUTsizesize of the file in bytes (integer)	7
	8 9
C binding	10
<pre>int MPI_File_get_size(MPI_File fh, MPI_Offset *size)</pre>	11
Fortran 2008 binding	12
MPI_File_get_size(fh, size, ierror)	13
TYPE(MPI_File), INTENT(IN) :: fh	14
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	16
Fortran binding	17
MPI_FILE_GET_SIZE(FH, SIZE, IERROR)	18
INTEGER FH, IERROR	19 20
INTEGER(KIND=MPI_OFFSET_KIND) SIZE	20
MPI_FILE_GET_SIZE returns, in size, the current size in bytes of the file associated with	22
the file handle fh. As far as consistency semantics are concerned, MPI_FILE_GET_SIZE is a	23
data access operation (see Section 14.6.1).	24
	25
14.2.7 Querying File Parameters	26
	27
	28
MPI_FILE_GET_GROUP(fh, group)	29
IN fh file handle (handle)	30
OUT group group which opened the file (handle)	31 32
group which opened the file (number)	33
C binding	34
int MPI_File_get_group(MPI_File fh, MPI_Group *group)	35
	36
Fortran 2008 binding	37
<pre>MPI_File_get_group(fh, group, ierror) TYPE(MPI_File), INTENT(IN) :: fh</pre>	38
TYPE(MPI_FILE), INTENT(IN) :: In TYPE(MPI_Group), INTENT(OUT) :: group	39
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
	41
Fortran binding	42
MPI_FILE_GET_GROUP(FH, GROUP, IERROR)	43 44
INTEGER FH, GROUP, IERROR	44 45
$MPI_FILE_GET_GROUP$ returns a duplicate of the group of the communicator used to	40
open the file associated with fh. The group is returned in group. The user is responsible for	47

freeing group.

```
1
     MPI_FILE_GET_AMODE(fh, amode)
2
       IN
                fh
                                            file handle (handle)
3
       OUT
                amode
                                            file access mode used to open the file (integer)
4
5
6
     C binding
7
     int MPI_File_get_amode(MPI_File fh, int *amode)
8
     Fortran 2008 binding
9
     MPI_File_get_amode(fh, amode, ierror)
10
         TYPE(MPI_File), INTENT(IN) :: fh
11
         INTEGER, INTENT(OUT) :: amode
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
     MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
15
16
         INTEGER FH, AMODE, IERROR
17
         MPI_FILE_GET_AMODE returns, in amode, the access mode of the file associated with
18
     fh.
19
20
     Example 14.1 In Fortran 77, decoding an amode bit vector will require a routine such as
21
     the following:
22
23
     SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
^{24}
     !
25
     !
         TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE
26
         IF SET, RETURN 1 IN BIT_FOUND, O OTHERWISE
     !
27
     ŗ
28
         INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
29
         BIT_FOUND = 0
30
         CP_AMODE = AMODE
^{31}
     100 CONTINUE
32
         LBIT = 0
33
         HIFOUND = 0
34
         DO L = MAX_BIT, 0, -1
35
             MATCHER = 2**L
36
             IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
37
                 HIFOUND = 1
38
                LBIT = MATCHER
39
                CP_AMODE = CP_AMODE - MATCHER
40
             END IF
41
         END DO
42
         IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
43
         IF (BIT_FOUND .EQ. O .AND. HIFOUND .EQ. 1 .AND. &
44
              CP_AMODE .GT. 0) GO TO 100
45
     END
46
47
         This routine could be called successively to decode amode, one bit at a time. For
```

⁴⁸ example, the following code fragment would check for MPI_MODE_RDONLY.

```
CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)
IF (BIT_FOUND .EQ. 1) THEN
PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE
ELSE
PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE
END IF
```

14.2.8 File Info

Hints specified via info (see Chapter 10) allow a user to provide information such as file access patterns and file system specifics to direct optimization. Providing hints may enable an implementation to deliver increased I/O performance or minimize the use of system resources. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI_FILE_GET_INFO) and that place a restriction on the behavior of the application. Hints are specified on a per file basis, in MPI_FILE_OPEN, MPI_FILE_DELETE, MPI_FILE_SET_VIEW, and MPI_FILE_SET_INFO, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46

MPI_FILE_SET_INFO updates the hints of the file associated with fh using the hints provided in info. This operation has no effect on previously set or defaulted hints that are not 48

 24

1 specified by info. It also has no effect on previously set or defaulted hints that are specified $\mathbf{2}$ by info, but are ignored by the MPI implementation in this call to MPI_FILE_SET_INFO. 3 MPI_FILE_SET_INFO is a collective routine. The info object may be different on each 4 process, but any info entries that an implementation requires to be the same on all processes 5must appear with the same value in each process's info object. 6

Advice to users. Many info items that an implementation can use when it creates or opens a file cannot easily be changed once the file has been created or opened. Thus, an implementation may ignore hints issued in this call that it would have accepted in an open call. An implementation may also be unable to update certain info hints in a call to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO. MPI_FILE_GET_INFO can be used to determine whether info changes were ignored by the implementation. (End of advice to users.)

file handle (handle)

new info object (handle)

16

 $\overline{7}$

8

9

10

11

12

13 1415

17

18 19

2021

22

2526

27

28

MPI_FILE_GET_INFO(fh, info_used) IN

OUT

C binding

```
int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
23
```

Fortran 2008 binding 24

fh

info_used

```
MPI_File_get_info(fh, info_used, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
```

TYPE(MPI_Info), INTENT(OUT) :: info_used

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

29Fortran binding

```
30
     MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
^{31}
         INTEGER FH, INFO_USED, IERROR
32
```

33 MPI_FILE_GET_INFO returns a new info object containing the hints of the file associ-34ated with fh. The current setting of all hints related to this file is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementa-35 36 tion and have default values specified; any user-supplied hints that were not ignored by the 37 implementation; and any additional hints that were set by the implementation. If no such 38hints exist, a handle to a newly created info object is returned that contains no key/value 39 pairs. The user is responsible for freeing info_used via MPI_INFO_FREE.

- 40
- 41 Reserved File Hints

42Some potentially useful hints (info key values) are outlined below. The following key values 43 are reserved. An implementation is not required to interpret these key values, but if it does 44interpret the key value, it must provide the functionality described. (For more details on 45"info," see Chapter 10.) 46

These hints mainly affect access patterns and the layout of data on parallel I/O devices. 47For each hint name introduced, we describe the purpose of the hint, and the type of the hint 48

Unofficial Draft for Comment Only

value. The "[**SAME**]" annotation specifies that the hint values provided by all participating processes must be identical; otherwise the program is erroneous. In addition, some hints are context dependent, and are only used by an implementation at specific times (e.g., "file_perm" is only useful during file creation).

- "access_style" (comma separated list of strings): This hint specifies the manner in which the file will be accessed until the file is closed or until the "access_style" key value is altered. The hint value is a comma separated list of the following: "read_once", "write_once", "read_mostly", "write_mostly", "sequential", "reverse_sequential", and "random".
- "collective_buffering" (boolean) [SAME]: This hint specifies whether the application may benefit from collective buffering. Collective buffering is an optimization performed on collective accesses. Accesses to the file are performed on behalf of all processes in the group by a number of target nodes. These target nodes coalesce small requests into large disk accesses. Valid values for this key are "true" and "false". Collective buffering parameters are further directed via additional hints: "cb_block_size", "cb_buffer_size", and "cb_nodes".
- "cb_block_size" (integer) [SAME]: This hint specifies the block size to be used for collective buffering file access. *Target nodes* access data in chunks of this size. The chunks are distributed among target nodes in a round-robin (cyclic) pattern.
- "cb_buffer_size" (integer) [SAME]: This hint specifies the total buffer space that can be used for collective buffering on each target node, usually a multiple of "cb_block_size".
- "cb_nodes" (integer) [SAME]: This hint specifies the number of target nodes to be used for collective buffering.
- "chunked" (comma separated list of integers) [SAME]: This hint specifies that the file consists of a multidimentional array that is often accessed by subarrays. The value for this hint is a comma separated list of array dimensions, starting from the most significant one (for an array stored in row-major order, as in C, the most significant dimension is the first one; for an array stored in column-major order, as in Fortran, the most significant dimension is the last one, and array dimensions should be reversed).
- "chunked_item" (comma separated list of integers) [SAME]: This hint specifies the size of each array entry, in bytes.
- "chunked_size" (comma separated list of integers) [SAME]: This hint specifies the dimensions of the subarrays. This is a comma separated list of array dimensions, starting from the most significant one.
- "filename" (string): This hint specifies the file name used when the file was opened. If the implementation is capable of returning the file name of an open file, it will be returned using this key by MPI_FILE_GET_INFO. This key is ignored when passed to MPI_FILE_OPEN, MPI_FILE_SET_VIEW, MPI_FILE_SET_INFO, and MPI_FILE_DELETE.
- "file_perm" (string) [SAME]: This hint specifies the file permissions to use for file creation. Setting this hint is only useful when passed to MPI_FILE_OPEN with an amode

Unofficial Draft for Comment Only

 $\overline{7}$

 31

1 2	that includes MPI_MODE_CREATE. The set of valid values for this key is implementa- tion dependent.			
3 4 5 6	"io_node_list" (comma separated list of strings) [SAME]: This hint specifies the list of I/O devices that should be used to store the file. This hint is most relevant when the file is created.			
7 8 9 10	"nb_proc" (integer) [SAME]: This hint specifies the number of parallel processes that will typically be assigned to run programs that access this file. This hint is most relevant when the file is created.			
11 12		· - ,	[SAME]: This hint specifies the number of I/O devices in the most relevant when the file is created.	
13 14 15		(e ,	(SAME): This hint specifies the number of I/O devices that iped across, and is relevant only when the file is created.	
16 17 18 19 20	used I/O	for this file. The device before pre-	[SAME]: This hint specifies the suggested striping unit to be e striping unit is the amount of consecutive data assigned to one cogressing to the next device, when striping across a number of sed in bytes. This hint is relevant only when the file is created.	
21 22 23 24	14.3 Fi	le Views		
25	MPI_FILE_	_SET_VIEW(fh,	disp, etype, filetype, datarep, info)	
26 27	INOUT	fh	file handle (handle)	
28	IN	disp	displacement (integer)	
29	IN	etype	elementary datatype (handle)	
30 31	IN	filetype	filetype (handle)	
32	IN	datarep	data representation (string)	
33	IN	info	info object (handle)	
34 35				
36	C binding	g		
37	int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,			
38		MPI_Data	type filetype, const char *datarep, MPI_Info info)	
39	Fortran 2	2008 binding		
40 41	MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)			
42	TYPE(MPI_File), INTENT(IN) :: fh			
43	INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype			
44	CHARACTER(LEN=*), INTENT(IN) :: datarep			
45	TYPE(MPI_Info), INTENT(IN) :: info			
46	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
47	Fortran b	oinding		
48				

MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER(KIND=MPI_OFFSET_KIND) DISP CHARACTER*(*) DATAREP

The MPI_FILE_SET_VIEW routine changes the process's view of the data in the file. The start of the view is set to disp; the type of data is set to etype; the distribution of data to processes is set to filetype; and the representation of data in the file is set to datarep. In addition, MPI_FILE_SET_VIEW resets the individual file pointers and the shared file pointer to zero. MPI_FILE_SET_VIEW is collective; the values for datarep and the extents of etype in the file data representation must be identical on all processes in the group; values for disp, filetype, and info may vary. The datatypes passed in etype and filetype must be committed.

The etype always specifies the data layout in the file. If etype is a portable datatype (see Section 2.4), the extent of etype is computed by scaling any displacements in the datatype to match the file data representation. If etype is not a portable datatype, no scaling is done when computing the extent of etype. The user must be careful when using nonportable etypes in heterogeneous environments; see Section 14.5.1 for further details.

If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, the special displacement MPI_DISPLACEMENT_CURRENT must be passed in disp. This sets the displacement to the current position of the shared file pointer. MPI_DISPLACEMENT_CURRENT is invalid unless the amode for the file has MPI_MODE_SEQUENTIAL set.

Rationale. For some sequential files, such as those corresponding to magnetic tapes or streaming network connections, the *displacement* may not be meaningful. MPI_DISPLACEMENT_CURRENT allows the view to be changed for these types of files. (*End of rationale.*)

Advice to implementors. It is expected that a call to MPI_FILE_SET_VIEW will immediately follow MPI_FILE_OPEN in numerous instances. A high-quality implementation will ensure that this behavior is efficient. (*End of advice to implementors.*)

The disp displacement argument specifies the position (absolute offset in bytes from the beginning of the file) where the view begins.

Advice to users. disp can be used to skip headers or when the file includes a sequence of data segments that are to be accessed in different patterns (see Figure 14.3). Separate views, each using a different displacement and filetype, can be used to access each segment.

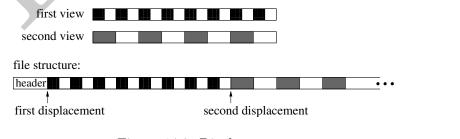


Figure 14.3: Displacements

(End of advice to users.)

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

 $45 \\ 46$

An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed by using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes.

> Advice to users. In order to ensure interoperability in a heterogeneous environment, additional restrictions must be observed when constructing the etype (see Section 14.5). (End of advice to users.)

A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. These displacements are not required to be distinct, but they cannot be negative, and they must be monotonically nondecreasing.

If the file is opened for writing, neither the etype nor the filetype is permitted to contain overlapping regions. This restriction is equivalent to the "datatype used in a receive cannot specify overlapping regions" restriction for communication. Note that filetypes from different processes may still overlap each other.

If a filetype has holes in it, then the data in the holes is inaccessible to the calling process. However, the disp, etype, and filetype arguments can be changed via future calls to MPI_FILE_SET_VIEW to access a different part of the file.

It is erroneous to use absolute addresses in the construction of the etype and filetype. The info argument is used to provide information regarding file access patterns and file system specifics to direct optimization (see Section 14.2.8). The constant MPI_INFO_NULL refers to the null info and can be used when no info needs to be specified.

The datarep argument is a string that specifies the representation of data in the file. See the file interoperability section (Section 14.5) for details and a discussion of valid values.

The user is responsible for ensuring that all nonblocking requests and split collective operations on fh have been completed before calling MPI_FILE_SET_VIEW — otherwise, the call to MPI_FILE_SET_VIEW is erroneous.

32 33

34

41 42 MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)

35	1N	fh	file handle (handle)
36	Ουτ	disp	displacement (integer)
37 38	OUT	etype	elementary datatype (handle)
39	OUT	filetype	filetype (handle)
40	OUT	datarep	data representation (string)
41			

C binding

```
46 Fortran 2008 binding
```

```
    MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
```

Unofficial Draft for Comment Only

7

8

9

10 11

12

13

14

```
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
CHARACTER(LEN=*), INTENT(OUT) :: datarep
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
INTEGER FH, ETYPE, FILETYPE, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) DISP
CHARACTER*(*) DATAREP
```

MPI_FILE_GET_VIEW returns the process's view of the data in the file. The current value of the displacement is returned in disp. The etype and filetype are new datatypes with typemaps equal to the typemaps of the current etype and filetype, respectively.

The data representation is returned in datarep. The user is responsible for ensuring that datarep is large enough to hold the returned data representation string. The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING.

In addition, if a portable datatype was used to set the current view, then the corresponding datatype returned by MPI_FILE_GET_VIEW is also a portable datatype. If etype or filetype are derived datatypes, the user is responsible for freeing them. The etype and filetype returned are both in a committed state.

14.4 Data Access

14.4.1 Data Access Routines

Data is moved between files and processes by issuing read and write calls. There are three orthogonal aspects to data access: positioning (explicit offset *vs.* implicit file pointer), synchronism (blocking *vs.* nonblocking and split collective), and coordination (noncollective *vs.* collective). The following combinations of these data access routines, including two types of file pointers (individual and shared) are provided in Table 14.1.

positioning	synchronism	coordination	
		noncollective	collective
explicit	blocking	MPI_FILE_READ_AT	MPI_FILE_READ_AT_ALL
offsets		MPI_FILE_WRITE_AT	MPI_FILE_WRITE_AT_ALL
	nonblocking	MPI_FILE_IREAD_AT	MPI_FILE_IREAD_AT_ALL
		MPI_FILE_IWRITE_AT	MPI_FILE_IWRITE_AT_ALL
	split collective	N/A	MPI_FILE_READ_AT_ALL_BEGIN
			MPI_FILE_READ_AT_ALL_END
			MPI_FILE_WRITE_AT_ALL_BEGIN
			MPI_FILE_WRITE_AT_ALL_END
individual	blocking	MPI_FILE_READ	MPI_FILE_READ_ALL
file pointers		MPI_FILE_WRITE	MPI_FILE_WRITE_ALL
	nonblocking	MPI_FILE_IREAD	MPI_FILE_IREAD_ALL
		MPI_FILE_IWRITE	MPI_FILE_IWRITE_ALL
	split collective	N/A	MPI_FILE_READ_ALL_BEGIN
			MPI_FILE_READ_ALL_END
			MPI_FILE_WRITE_ALL_BEGIN
			MPI_FILE_WRITE_ALL_END
shared	blocking	MPI_FILE_READ_SHARED	MPI_FILE_READ_ORDERED
file pointer		MPI_FILE_WRITE_SHARED	MPI_FILE_WRITE_ORDERED
	nonblocking	MPI_FILE_IREAD_SHARED	N/A
		MPI_FILE_IWRITE_SHARED	,
	split collective	N/A	MPI_FILE_READ_ORDERED_BEGIN
		· ·	MPI_FILE_READ_ORDERED_END
			MPI_FILE_WRITE_ORDERED_BEGIN
			MPI_FILE_WRITE_ORDERED_END
		1	

Table 14.1: Data access routines

Unofficial Draft for Comment Only

 $\mathbf{2}$

POSIX read()/fread() and write()/fwrite() are blocking, noncollective operations and use individual file pointers. The MPI equivalents are MPI_FILE_READ and 3 MPI_FILE_WRITE. 4 Implementations of data access routines may buffer data to improve performance. This does not affect reads, as the data is always available in the user's buffer after a read operation 6 completes. For writes, however, the MPI_FILE_SYNC routine provides the only guarantee that data has been transferred to the storage device.

9 Positioning 10

MPI provides three types of positioning for data access routines: explicit offsets, indi-11 vidual file pointers, and shared file pointers. The different positioning methods may 12be mixed within the same program and do not affect each other. 13

The data access routines that accept explicit offsets contain _AT in their name (e.g., 14MPI_FILE_WRITE_AT). Explicit offset operations perform data access at the file position 15given directly as an argument — no file pointer is used nor updated. Note that this is not 16equivalent to an atomic seek-and-read or seek-and-write operation, as no "seek" is issued. 17Operations with explicit offsets are described in Section 14.4.2. 18

The names of the individual file pointer routines contain no positional qualifier (e.g., 19MPI_FILE_WRITE). Operations with individual file pointers are described in Section 14.4.3. 20The data access routines that use shared file pointers contain _SHARED or _ORDERED 21in their name (e.g., MPI_FILE_WRITE_SHARED). Operations with shared file pointers are 22 described in Section 14.4.4. 23

The main semantic issues with MPI-maintained file pointers are how and when they are 24 updated by I/O operations. In general, each I/O operation leaves the file pointer pointing to 25the next data item after the last one that is accessed by the operation. In a nonblocking or 26split collective operation, the pointer is updated by the call that initiates the I/O, possibly 27before the access completes. 28

More formally,

29 30 31

32

37

 $new_file_offset = old_file_offset + \frac{elements(datatype)}{elements(etupe)} \times count$

where *count* is the number of *datatype* items to be accessed, elements(X) is the number of 33 predefined datatypes in the typemap of X, and old_file_offset is the value of the implicit 34offset before the call. The file position, *new_file_offset*, is in terms of a count of etypes 35 relative to the current view. 36

Synchronism 38

39 MPI supports blocking and nonblocking I/O routines. 40

A blocking I/O call will not return until the I/O request is completed.

41 A nonblocking I/O call initiates an I/O operation, but does not wait for it to complete. 42Given suitable hardware, this allows the transfer of data out of and into the user's buffer 43 to proceed concurrently with computation. A separate request complete call (MPI_WAIT, 44MPI_TEST, or any of their variants) is needed to complete the I/O request, i.e., to confirm 45that the data has been read or written and that it is safe for the user to reuse the buffer. 46 The nonblocking versions of the routines are named MPI_FILE_IXXX, where the I stands 47for immediate. 48

1

 $\mathbf{2}$

5

 $\overline{7}$

 $\mathbf{2}$

 24

 31

It is erroneous to access the local buffer of a nonblocking data access operation, or to use that buffer as the source or target of other communications, between the initiation and completion of the operation.

The split collective routines support a restricted form of "nonblocking" operations for collective data access (see Section 14.4.5).

Coordination

Every noncollective data access routine MPI_FILE_XXX has a collective counterpart. For most routines, this counterpart is MPI_FILE_XXX_ALL or a pair of MPI_FILE_XXX_BEGIN and MPI_FILE_XXX_END. The counterparts to the MPI_FILE_XXX_SHARED routines are MPI_FILE_XXX_ORDERED.

The completion of a noncollective call only depends on the activity of the calling process. However, the completion of a collective call (which must be called by all members of the process group) may depend on the activity of the other processes participating in the collective call. See Section 14.6.4 for rules on semantics of collective calls.

Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimization.

Data Access Conventions

Data is moved between files and processes by calling read and write routines. Read routines move data from a file into memory. Write routines move data from memory into a file. The file is designated by a file handle, fh. The location of the file data is specified by an offset into the current view. The data in memory is specified by a triple: buf, count, and datatype. Upon completion, the amount of data accessed by the calling process is returned in a status.

An offset designates the starting position in the file for an access. The offset is always in etype units relative to the current view. Explicit offset routines pass offset as an argument (negative values are erroneous). The file pointer routines use implicit offsets maintained by MPI.

A data access routine attempts to transfer (read or write) count data items of type datatype between the user's buffer buf and the file. The datatype passed to the routine must be a committed datatype. The layout of data in memory corresponding to buf, count, datatype is interpreted the same way as in MPI communication functions; see Section 3.2.2 and Section 5.1.11. The data is accessed from those parts of the file specified by the current view (Section 14.3). The type signature of datatype must match the type signature of some number of contiguous copies of the etype of the current view. As in a receive, it is erroneous to specify a datatype for reading that contains overlapping regions (areas of memory which would be stored into more than once).

The nonblocking data access routines indicate that MPI can start a data access and associate a request handle, request, with the I/O operation. Nonblocking operations are completed via MPI_TEST, MPI_WAIT, or any of their variants.

Data access operations, when completed, return the amount of data accessed in status.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 19.1.10–19.1.20. (End of advice to users.)

1 For blocking routines, status is returned directly. For nonblocking routines and split $\mathbf{2}$ collective routines, status is returned when the operation is completed. The number of 3 datatype entries and predefined elements accessed by the calling process can be extracted 4 from status by using MPI_GET_COUNT and MPI_GET_ELEMENTS (or $\mathbf{5}$ MPI_GET_ELEMENTS_X), respectively. The interpretation of the MPI_ERROR field is the 6 same as for other operations — normally undefined, but meaningful if an MPI routine 7returns MPI_ERR_IN_STATUS. The user can pass (in C and Fortran) MPI_STATUS_IGNORE 8 in the status argument if the return value of this argument is not needed. The status can be 9 passed to MPI_TEST_CANCELLED to determine if the operation was cancelled. All other 10 fields of status are undefined. 11When reading, a program can detect the end of file by noting that the amount of data 12read is less than the amount requested. Writing past the end of file increases the file size. 13 The amount of data accessed will be the amount requested, unless an error is raised (or a 14read reaches the end of file). 151614.4.2 Data Access with Explicit Offsets 17If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to 18 call the routines in this section. 19

```
MPI_FILE_READ_AT(fh, offset, buf, count, datatype, status)
```

22		,	,
23	IN	fh	file handle (handle)
24	IN	offset	file offset (integer)
25 26	OUT	buf	initial address of buffer (choice)
27	IN	count	number of elements in buffer (integer)
28	IN	datatype	datatype of each buffer element (handle)
29 30	OUT	status	status object (Status)
31			

```
C binding
```

20 21

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
```

```
<sup>35</sup> Fortran 2008 binding
```

```
<sup>36</sup> MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
<sup>37</sup> TYPE(MPI_File), INTENT(IN) :: fh
<sup>38</sup> INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
<sup>39</sup> TYPE(*) DIMENSION(*) :: buf
```

```
<sup>39</sup> TYPE(*), DIMENSION(..) :: buf
```

```
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

```
<sup>42</sup> TYPE(MPI_Status) :: status
```

```
<sup>43</sup> INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
44
45 Fortran binding
```

```
    MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

<type< td=""><td>e> BUF(*)</td><td></td><td>1</td></type<>	e> BUF(*)		1
MPI_FILE_READ_AT reads a file beginning at the position specified by offset.			2 3
_			3
	_READ_AT_ALL(fh, offset,	auf count datatyne status)	5
IN	fh	file handle (handle)	6
			7
IN	offset	file offset (integer)	8 9
OUT	buf	initial address of buffer (choice)	10
IN	count	number of elements in buffer (integer)	11
IN	datatype	datatype of each buffer element (handle)	12
OUT	status	status object (Status)	13
			14 15
C bindin	-		16
int MPI_		le fh, MPI_Offset offset, void *buf, atype datatype, MPI_Status *status)	17
		atype datatype, MI_Status *status/	18
	2008 binding	, buf, count, datatype, status, ierror)	19 20
	(MPI_File), INTENT(IN)		21
		D), INTENT(IN) :: offset	22
TYPE	(*), DIMENSION() :: b	uf	23
	GER, INTENT(IN) :: coun		24
	(MPI_Datatype), INTENT((MPI_Status) :: status	IN) :: datatype	25 26
	GER, OPTIONAL, INTENT(O	UT) :: ierror	27
			28
Fortran		, BUF, COUNT, DATATYPE, STATUS, IERROR)	29
		, STATUS(MPI_STATUS_SIZE), IERROR	30
	GER(KIND=MPI_OFFSET_KIN		31 32
<typ< td=""><td>e> BUF(*)</td><td></td><td>33</td></typ<>	e> BUF(*)		33
MPI_	FILE_READ_AT_ALL is a co	blective version of the blocking MPI_FILE_READ_AT	34
interface.		0	35
			36
MPI FILE	_WRITE_AT(fh, offset, buf,	count. datatype. status)	37 38
- INOUT	fh	file handle (handle)	39
IN	offset	file offset (integer)	40
			41
IN	buf	initial address of buffer (choice)	42
IN	count	number of elements in buffer (integer)	43 44
IN	datatype	datatype of each buffer element (handle)	45
OUT	status	status object (Status)	46
			47

C binding

```
1
     int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,
\mathbf{2}
                    int count, MPI_Datatype datatype, MPI_Status *status)
3
     Fortran 2008 binding
4
     MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
5
         TYPE(MPI_File), INTENT(IN) :: fh
6
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
7
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
8
         INTEGER, INTENT(IN) :: count
9
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
         TYPE(MPI_Status) :: status
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
     Fortran binding
14
     MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
15
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
16
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
17
         <type> BUF(*)
18
         MPI_FILE_WRITE_AT writes a file beginning at the position specified by offset.
19
20
21
     MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status)
22
       INOUT
                fh
                                           file handle (handle)
23
^{24}
       IN
                offset
                                           file offset (integer)
25
       IN
                buf
                                           initial address of buffer (choice)
26
       IN
                count
                                           number of elements in buffer (integer)
27
       IN
28
                datatype
                                           datatype of each buffer element (handle)
29
       OUT
                status
                                           status object (Status)
30
^{31}
     C binding
32
     int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,
33
                    int count, MPI_Datatype datatype, MPI_Status *status)
34
35
     Fortran 2008 binding
36
     MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
37
         TYPE(MPI_File), INTENT(IN) :: fh
38
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
39
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
40
         INTEGER, INTENT(IN) :: count
41
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
         TYPE(MPI_Status) :: status
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     Fortran binding
45
     MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
46
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
47
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
48
```

	<type> BUF(*)</type>		1
	MPI_FILE_WRITE_AT_A	ALL is a collective version of the blocking	2 3
MPI_	FILE_WRITE_AT interfa	ice.	4
			5
MPI_	FILE_IREAD_AT(fh, offs	et, buf, count, datatype, request)	6
IN	fh	file handle (handle)	7
IN	offset	file offset (integer)	8 9
OU		initial address of buffer (choice)	10
			11
IN	count	number of elements in buffer (integer)	12
IN	datatype	datatype of each buffer element (handle)	13 14
OU	T request	request object (handle)	14
C L:			16
	nding MPI File iread at(MP	I_File fh, MPI_Offset offset, void *buf, int count,	17
1110		e datatype, MPI_Request *request)	18
Fort	ran 2008 binding		19 20
	0	fset, buf, count, datatype, request, ierror)	21
	TYPE(MPI_File), INTEN		22
		SET_KIND), INTENT(IN) :: offset	23
		.), ASYNCHRONOUS :: buf	24
	INTEGER, INTENT(IN)	:: count INTENT(IN) :: datatype	25 26
		NTENT(OUT) :: request	27
	INTEGER, OPTIONAL, IN	-	28
Fort	ran binding		29
	Ŭ	FSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)	30
		ATATYPE, REQUEST, IERROR	31 32
	INTEGER(KIND=MPI_OFFS	SET_KIND) OFFSET	33
	<type> BUF(*)</type>		34
	$MPI_FILE_IREAD_AT \text{ is}$	a nonblocking version of the MPI_FILE_READ_AT interface.	35
			36
MPI	FILE_IREAD_AT_ALL(fr	n, offset, buf, count, datatype, request)	37 38
IN	fh	file handle (handle)	39
IN	offset	file offset (integer)	40
			41
OU		initial address of buffer (choice)	42
IN	count	number of elements in buffer (integer)	43 44
IN	datatype	datatype of each buffer element (handle)	45
OU	T request	request object (handle)	46
			47

```
1
     int MPI_File_iread_at_all(MPI_File fh, MPI_Offset offset, void *buf,
\mathbf{2}
                    int count, MPI_Datatype datatype, MPI_Request *request)
3
     Fortran 2008 binding
4
     MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror)
5
         TYPE(MPI_File), INTENT(IN) :: fh
6
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
8
         INTEGER, INTENT(IN) :: count
9
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
         TYPE(MPI_Request), INTENT(OUT) :: request
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
     Fortran binding
14
     MPI_FILE_IREAD_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
15
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
16
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
17
         <type> BUF(*)
18
         MPI_FILE_IREAD_AT_ALL is a nonblocking version of MPI_FILE_READ_AT_ALL. See
19
     Section 14.6.5 for semantics of nonblocking collective file operations.
20
21
22
     MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)
23
       INOUT
                fh
                                           file handle (handle)
24
                                           file offset (integer)
25
       IN
                offset
26
                                           initial address of buffer (choice)
       IN
                buf
27
                                           number of elements in buffer (integer)
       IN
                count
28
29
       IN
                datatype
                                           datatype of each buffer element (handle)
30
       OUT
                request
                                           request object (handle)
^{31}
32
     C binding
33
     int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,
34
                    int count, MPI_Datatype datatype, MPI_Request *request)
35
36
     Fortran 2008 binding
37
     MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
38
         TYPE(MPI_File), INTENT(IN) :: fh
39
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
40
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
41
         INTEGER, INTENT(IN) :: count
42
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
         TYPE(MPI_Request), INTENT(OUT) :: request
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     Fortran binding
46
     MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
47
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
48
```

<type< th=""><th>GER(KIND=MPI_OFFSET_KIND) >> BUF(*) FILE_IWRITE_AT is a nonblock</th><th>OFFSET king version of the MPI_FILE_WRITE_AT interface.</th><th>1 2 3 4 5</th></type<>	GER(KIND=MPI_OFFSET_KIND) >> BUF(*) FILE_IWRITE_AT is a nonblock	OFFSET king version of the MPI_FILE_WRITE_AT interface.	1 2 3 4 5
MPI_FILE	_IWRITE_AT_ALL(fh, offset, b	uf, count, datatype, request)	6
INOUT	fh	file handle (handle)	7
IN	offset	file offset (integer)	9
IN	buf	initial address of buffer (choice)	10
IN	count	number of elements in buffer (integer)	11
			12
IN	datatype	datatype of each buffer element (handle)	13 14
OUT	request	request object (handle)	15
C bindin	ar an		16
	-	le fh, MPI_Offset offset, const void *buf,	17
		ype datatype, MPI_Request *request)	18
Fortron 2	2008 binding		19 20
	0	, buf, count, datatype, request, ierror)	21
	(MPI_File), INTENT(IN) ::		22
	ER(KIND=MPI_OFFSET_KIND)		23
TYPE	(*), DIMENSION(), INTEN	I(IN), ASYNCHRONOUS :: buf	24
INTEGER, INTENT(IN) :: count			25
TYPE(MPI_Datatype), INTENT(IN) :: datatype		26 27	
TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror		27	
			29
Fortran k	3		30
		, BUF, COUNT, DATATYPE, REQUEST, IERROR)	31
	ER FH, COUNT, DATATYPE, H ER(KIND=MPI_OFFSET_KIND)		32
	<pre>BUF(*)</pre>		33
			34 35
MPI_I	-ILE_IVVRITE_AT_ALL is a no	nblocking version of MPI_FILE_WRITE_AT_ALL.	36
14.4.3 D	ata Access with Individual Fi	le Pointers	37
			38
	-	er per process per file handle. The current value	39
-		ffset in the data access routines described in this late the individual file pointers maintained by MPI.	40
	d file pointer is not used nor u	* *	41
	-	have the same semantics as the data access with	42 43
	_	ion 14.4.2, with the following modification:	44
+ the	offect is defined to be the	mont value of the MDI maintained individual fil-	45
• the opin		rrent value of the MPI-maintained individual file	46
pom			47
			48

```
1
     After an individual file pointer operation is initiated, the individual file pointer is updated
\mathbf{2}
     to point to the next etype after the last one that will be accessed. The file pointer is updated
3
     relative to the current view of the file.
4
          If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous
5
     to call the routines in this section, with the exception of MPI_FILE_GET_BYTE_OFFSET.
6
7
     MPI_FILE_READ(fh, buf, count, datatype, status)
8
9
       INOUT
                 fh
                                              file handle (handle)
10
       OUT
                 buf
                                             initial address of buffer (choice)
11
                                              number of elements in buffer (integer)
       IN
                 count
12
13
                                             datatype of each buffer element (handle)
       IN
                 datatype
14
       OUT
                                              status object (Status)
                 status
15
16
     C binding
17
     int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
18
                     MPI_Status *status)
19
20
     Fortran 2008 binding
21
     MPI_File_read(fh, buf, count, datatype, status, ierror)
22
          TYPE(MPI_File), INTENT(IN) :: fh
23
          TYPE(*), DIMENSION(..) :: buf
24
          INTEGER, INTENT(IN) :: count
25
          TYPE(MPI_Datatype), INTENT(IN)
                                                 datatype
                                              ::
26
          TYPE(MPI_Status) :: status
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     Fortran binding
29
     MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
30
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
^{31}
          <type> BUF(*)
32
33
          MPI_FILE_READ reads a file using the individual file pointer.
34
     Example 14.2 The following Fortran code fragment is an example of reading a file until
35
     the end of file is reached:
36
37
          Read a preexisting input file until all data has been read.
38
          Call routine "process_input" if all requested data is read.
     !
39
          The Fortran 90 "exit" statement exits the loop.
40
     Ţ
41
                 bufsize, numread, totprocessed, status(MPI_STATUS_SIZE)
42
     integer
     parameter (bufsize=100)
43
     real
                 localbuffer(bufsize)
44
     integer (kind=MPI_OFFSET_KIND) zero
45
46
47
     zero = 0
48
```

```
1
call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
                                                                                       \mathbf{2}
                    MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
call MPI_FILE_SET_VIEW(myfh, zero, MPI_REAL, MPI_REAL, 'native', &
                         MPI_INFO_NULL, ierr)
totprocessed = 0
do
   call MPI_FILE_READ(myfh, localbuffer, bufsize, MPI_REAL, &
                        status, ierr)
   call MPI_GET_COUNT(status, MPI_REAL, numread, ierr)
                                                                                       10
   call process_input(localbuffer, numread)
                                                                                       11
   totprocessed = totprocessed + numread
   if (numread < bufsize) exit
                                                                                       12
                                                                                       13
end do
                                                                                       14
                                                                                       15
write(6, 1001) numread, bufsize, totprocessed
1001 format("No more data: read", I3, "and expected", I3, &
                                                                                       16
                                                                                       17
              "Processed total of", I6, "before terminating job.")
                                                                                       18
                                                                                       19
call MPI_FILE_CLOSE(myfh, ierr)
                                                                                       20
                                                                                       21
                                                                                       22
MPI_FILE_READ_ALL(fh, buf, count, datatype, status)
                                                                                       23
 INOUT
           fh
                                      file handle (handle)
                                                                                       24
                                                                                       25
 OUT
           buf
                                      initial address of buffer (choice)
                                                                                       26
 IN
           count
                                      number of elements in buffer (integer)
                                                                                       27
 IN
                                      datatype of each buffer element (handle)
           datatype
                                                                                       28
                                                                                       29
 OUT
                                      status object (Status)
           status
                                                                                       30
                                                                                       31
C binding
                                                                                       32
int MPI_File_read_all(MPI_File fh, void *buf, int count,
                                                                                       33
              MPI_Datatype datatype, MPI_Status *status)
                                                                                       34
Fortran 2008 binding
                                                                                       35
MPI_File_read_all(fh, buf, count, datatype, status, ierror)
                                                                                       36
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                       37
    TYPE(*), DIMENSION(..) :: buf
                                                                                       38
    INTEGER, INTENT(IN) :: count
                                                                                       39
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                       40
                                                                                       41
    TYPE(MPI_Status) :: status
                                                                                       42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       43
Fortran binding
                                                                                       44
MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
                                                                                       45
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
                                                                                       46
    <type> BUF(*)
                                                                                       47
                                                                                       48
    MPI_FILE_READ_ALL is a collective version of the blocking MPI_FILE_READ interface.
```

```
1
     MPI_FILE_WRITE(fh, buf, count, datatype, status)
2
       INOUT
                 fh
                                             file handle (handle)
3
       IN
                 buf
                                             initial address of buffer (choice)
4
5
       IN
                 count
                                             number of elements in buffer (integer)
6
       IN
                                             datatype of each buffer element (handle)
                 datatype
7
       OUT
                 status
                                             status object (Status)
8
9
     C binding
10
11
     int MPI_File_write(MPI_File fh, const void *buf, int count,
                    MPI_Datatype datatype, MPI_Status *status)
12
13
     Fortran 2008 binding
14
     MPI_File_write(fh, buf, count, datatype, status, ierror)
15
          TYPE(MPI_File), INTENT(IN) :: fh
16
          TYPE(*), DIMENSION(...), INTENT(IN) :: buf
17
          INTEGER, INTENT(IN) :: count
18
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
          TYPE(MPI_Status) :: status
20
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     Fortran binding
23
     MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
24
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
25
          <type> BUF(*)
26
          MPI_FILE_WRITE writes a file using the individual file pointer.
27
28
29
     MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)
30
       INOUT
                 fh
                                             file handle (handle)
^{31}
       IN
                 buf
                                             initial address of buffer (choice)
32
33
       IN
                                             number of elements in buffer (integer)
                 count
34
       IN
                                             datatype of each buffer element (handle)
                 datatype
35
       OUT
                                             status object (Status)
                 status
36
37
38
     C binding
39
     int MPI_File_write_all(MPI_File fh, const void *buf, int count,
40
                    MPI_Datatype datatype, MPI_Status *status)
41
     Fortran 2008 binding
42
     MPI_File_write_all(fh, buf, count, datatype, status, ierror)
43
          TYPE(MPI_File), INTENT(IN) :: fh
44
          TYPE(*), DIMENSION(..), INTENT(IN) :: buf
45
          INTEGER, INTENT(IN) :: count
46
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
47
          TYPE(MPI_Status) :: status
48
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        1
                                                                                        2
Fortran binding
MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    <type> BUF(*)
                                                                                        6
    MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE inter-
                                                                                        7
face.
                                                                                        9
                                                                                        10
MPI_FILE_IREAD(fh, buf, count, datatype, request)
                                                                                        11
 INOUT
                                                                                        12
           fh
                                      file handle (handle)
                                                                                        13
 OUT
           buf
                                      initial address of buffer (choice)
                                                                                        14
 IN
           count
                                      number of elements in buffer (integer)
                                                                                        15
                                                                                        16
                                      datatype of each buffer element (handle).
 IN
           datatype
                                                                                        17
 OUT
                                      request object (handle)
           request
                                                                                        18
                                                                                        19
C binding
                                                                                        20
int MPI_File_iread(MPI_File fh, void *buf, int count,
                                                                                       21
              MPI_Datatype datatype, MPI_Request *request)
                                                                                        22
                                                                                        23
Fortran 2008 binding
                                                                                        24
MPI_File_iread(fh, buf, count, datatype, request, ierror)
                                                                                        25
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                        26
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                        27
    INTEGER, INTENT(IN) :: count
                                                                                        28
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                        29
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                        30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        31
Fortran binding
                                                                                        32
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
                                                                                        33
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
                                                                                       34
    <type> BUF(*)
                                                                                       35
                                                                                       36
    MPI_FILE_IREAD is a nonblocking version of the MPI_FILE_READ interface.
                                                                                       37
                                                                                        38
Example 14.3 The following Fortran code fragment illustrates file pointer update seman-
                                                                                        39
tics:
                                                                                        40
    Read the first twenty real words in a file into two local
                                                                                        41
!
!
    buffers. Note that when the first MPI_FILE_IREAD returns,
                                                                                        42
!
    the file pointer has been updated to point to the
                                                                                        43
    eleventh real word in the file.
!
                                                                                        44
                                                                                        45
          bufsize, req1, req2
integer
                                                                                        46
integer, dimension(MPI_STATUS_SIZE) :: status1, status2
                                                                                        47
parameter (bufsize=10)
                                                                                        48
```

```
1
     real
                buf1(bufsize), buf2(bufsize)
\mathbf{2}
     integer (kind=MPI_OFFSET_KIND) zero
3
4
     zero = 0
5
     call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
6
                          MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
7
     call MPI_FILE_SET_VIEW(myfh, zero, MPI_REAL, MPI_REAL, 'native', &
8
                               MPI_INFO_NULL, ierr)
9
     call MPI_FILE_IREAD(myfh, buf1, bufsize, MPI_REAL, &
10
                           req1, ierr)
^{11}
     call MPI_FILE_IREAD(myfh, buf2, bufsize, MPI_REAL, &
12
                           req2, ierr)
13
14
     call MPI_WAIT(req1, status1, ierr)
15
     call MPI_WAIT(req2, status2, ierr)
16
17
     call MPI_FILE_CLOSE(myfh, ierr)
18
19
20
     MPI_FILE_IREAD_ALL(fh, buf, count, datatype, request)
21
22
       INOUT
                fh
                                            file handle (handle)
23
       OUT
                buf
                                            initial address of buffer (choice)
24
       IN
                                            number of elements in buffer (integer)
                count
25
26
       IN
                                            datatype of each buffer element (handle)
                datatype
27
       OUT
                request
                                            request object (handle)
28
29
     C binding
30
     int MPI_File_iread_all(MPI_File fh, void *buf, int count,
^{31}
                    MPI_Datatype datatype, MPI_Request *request)
32
33
     Fortran 2008 binding
34
     MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
35
         TYPE(MPI_File), INTENT(IN) :: fh
36
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
37
         INTEGER, INTENT(IN) :: count
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         TYPE(MPI_Request), INTENT(OUT) :: request
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
     Fortran binding
42
     MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
43
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
44
          <type> BUF(*)
45
46
         MPI_FILE_IREAD_ALL is a nonblocking version of MPI_FILE_READ_ALL.
47
48
```

MPI_FILE_IWRITE(fh, buf, count, datatype, request)			1
INOUT	fh	file handle (handle)	2
IN	buf	initial address of buffer (choice)	$\frac{3}{4}$
IN	count	number of elements in buffer (integer)	5
IN	datatype	datatype of each buffer element (handle)	6
	51	·-	7
OUT	request	request object (handle)	8
C bindin	α.		9 10
	-	const void *buf, int count,	10
1110 111 1_1		e, MPI_Request *request)	12
Dentanan (13
	2008 binding _iwrite(fh, buf, count, da	atatuna request jarror)	14
	(MPI_File), INTENT(IN) ::		15
		Γ(IN), ASYNCHRONOUS :: buf	16
INTEC	GER, INTENT(IN) :: count		17 18
	(MPI_Datatype), INTENT(IN)		19
	(MPI_Request), INTENT(OUT)	-	20
INTEC	SER, OPTIONAL, INTENT(OUT)) :: lerror	21
Fortran binding			22
MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)			23
	GER FH, COUNT, DATATYPE, I	REQUEST, IERROR	24 25
<type< td=""><td>e> BUF(*)</td><td></td><td>25</td></type<>	e> BUF(*)		25
MPI_I	FILE_IWRITE is a nonblocking	g version of the MPI_FILE_WRITE interface.	27
			28
MPI_FILE	_IWRITE_ALL(fh, buf, count, c	datatype, request)	29
INOUT	fh	file handle (handle)	30
IN	buf	initial address of buffer (choice)	31 32
			33
IN	count	number of elements in buffer (integer)	34
IN	datatype	datatype of each buffer element (handle)	35
OUT	request	request object (handle)	36
			37
C bindin	-		38 30
int MPI_F		fh, const void *buf, int count,	39 40
	MPI_Datatype datatyp	e, MPI_Request *request)	41
Fortran 2	2008 binding		10

```
Fortran 2008 binding
MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
```

42

43

44

45

46

47

48

```
1
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     Fortran binding
3
     MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
4
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
5
          <type> BUF(*)
6
7
          MPI_FILE_IWRITE_ALL is a nonblocking version of MPI_FILE_WRITE_ALL.
8
9
     MPI_FILE_SEEK(fh, offset, whence)
10
11
       INOUT
                 fh
                                              file handle (handle)
12
       IN
                 offset
                                              file offset (integer)
13
       IN
                 whence
                                              update mode (state)
14
15
16
     C binding
17
     int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
18
     Fortran 2008 binding
19
     MPI_File_seek(fh, offset, whence, ierror)
20
          TYPE(MPI_File), INTENT(IN) :: fh
21
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
22
          INTEGER, INTENT(IN) :: whence
23
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
25
     Fortran binding
26
     MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
27
          INTEGER FH, WHENCE, IERROR
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
28
29
          MPI_FILE_SEEK updates the individual file pointer according to whence, which has the
30
     following possible values:
^{31}
32
         • MPI_SEEK_SET: the pointer is set to offset
33
         • MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
34
35
         • MPI_SEEK_END: the pointer is set to the end of file plus offset
36
37
          The offset can be negative, which allows seeking backwards. It is erroneous to seek to
     a negative position in the view.
38
39
40
     MPI_FILE_GET_POSITION(fh, offset)
41
42
       IN
                 fh
                                              file handle (handle)
43
       OUT
                 offset
                                              offset of individual pointer (integer)
44
45
     C binding
46
     int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
47
48
     Fortran 2008 binding
```

MPI_File_get_position(fh, offset, ierror)	1
TYPE(MPI_File), INTENT(IN) :: fh	2
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
Fortran binding	5
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)	6
INTEGER FH, IERROR	7 8
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	9
MPI_FILE_GET_POSITION returns, in offset, the current position of the individual file	10
pointer in etype units relative to the current view.	11
pointer in coppe and relative to the carrent view.	12
Advice to users. The offset can be used in a future call to MPI_FILE_SEEK using	13
whence = MPI_SEEK_SET to return to the current position. To set the displacement to	14
the current file pointer position, first convert offset into an absolute byte position using	15
MPI_FILE_GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with the resulting	16
displacement. (End of advice to users.)	17
	18 19
	20
MPI_FILE_GET_BYTE_OFFSET(fh, offset, disp)	21
IN fh file handle (handle)	22
IN offset offset (integer)	23
	24
OUTdispabsolute byte position of offset (integer)	25
Chinding	26
C binding int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,	27
MPI_Offset *disp)	28 29
	30
Fortran 2008 binding	31
MPI_File_get_byte_offset(fh, offset, disp, ierror)	32
TYPE(MPI_File), INTENT(IN) :: fh	33
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp	34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	35
	36
Fortran binding	37
MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)	38
INTEGER FH, IERROR	39
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP	40 41
$MPI_FILE_GET_BYTE_OFFSET$ converts a view-relative offset into an absolute by te	41
position. The absolute byte position (from the beginning of the file) of offset relative to the	43
current view of fh is returned in disp.	44

14.4.4 Data Access with Shared File Pointers

MPI maintains exactly one shared file pointer per collective MPI_FILE_OPEN (shared among processes in the communicator group). The current value of this pointer implicitly specifies 48

Unofficial Draft for Comment Only

45

1 2	update the	e shared file point	routines described in this section. These routines only use and er maintained by MPI. The individual file pointers are not used
3 4 5		nared file pointer i	coutines have the same semantics as the data access with explicit Section 14.4.2, with the following modifications:
6			be the current value of the MPI-maintained shared file pointer,
7 8 9	• the e		calls to shared file pointer routines is defined to behave as if the
10 11 12	• the u file v		pointer routines is erroneous unless all processes use the same
13 14 15 16 17 18	istic. The After point to th	user needs to use a shared file poin	file pointer routines, the serialization ordering is not determin- other synchronization means to enforce a specific order. Iter operation is initiated, the shared file pointer is updated to r the last one that will be accessed. The file pointer is updated of the file.
19 20 21	Noncollecti	ve Operations	
22 23	MPI_FILE_	_READ_SHARED((fh, buf, count, datatype, status)
24	INOUT	fh	file handle (handle)
25	OUT	buf	initial address of buffer (choice)
26 27	IN	count	number of elements in buffer (integer)
28	IN	datatype	datatype of each buffer element (handle)
29 30	OUT	status	status object (Status)
31 32 33 34		ile_read_share MPI_Dataty	d(MPI_File fh, void *buf, int count, ppe datatype, MPI_Status *status)
35 36		2008 binding	, buf, count, datatype, status, ierror)
37		MPI_File), INT	• -
38		*), DIMENSION(
39		ER, INTENT(IN)	
40 41		MPI_Datatype), MPI_Status) ::	INTENT(IN) :: datatype
42			INTENT(OUT) :: ierror
43 44 45 46 47	INTEG	READ_SHARED(FH	, BUF, COUNT, DATATYPE, STATUS, IERROR) DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
48	MPI_F	FILE_READ_SHAI	RED reads a file using the shared file pointer.

MPI_FILE_	WRITE_SHARED(fh, buf, cou	nt, datatype, status)	1
INOUT	fh	file handle (handle)	2
IN	buf	initial address of buffer (choice)	$\frac{3}{4}$
IN	count	number of elements in buffer (integer)	5
IN	datatype	datatype of each buffer element (handle)	6
OUT	status	status object (Status)	7
001	Status	Status object (Status)	8 9
C binding	5		10
int MPI_F	ile_write_shared(MPI_File	e fh, const void *buf, int count,	11
	MPI_Datatype datatype	e, MPI_Status *status)	12
Fortran 2	008 binding		13 14
		nt, datatype, status, ierror)	15
	MPI_File), INTENT(IN) :: *), DIMENSION(), INTENT		16
	ER, INTENT(IN) :: count	(IN) Bul	17
	MPI_Datatype), INTENT(IN)	:: datatype	18 19
	MPI_Status) :: status		20
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	21
Fortran b	inding		22
		NT, DATATYPE, STATUS, IERROR)	23
		TATUS(MPI_STATUS_SIZE), IERROR	24 25
• -	> BUF(*)		26
MPI_F	ILE_WRITE_SHARED writes	a file using the shared file pointer.	27
			28
MPI_FILE_	IREAD_SHARED(fh, buf, cour	nt, datatype, request)	29 30
INOUT	fh	file handle (handle)	31
OUT	buf	initial address of buffer (choice)	32
IN	count	number of elements in buffer (integer)	33
IN	datatype	datatype of each buffer element (handle)	34 35
OUT	request	request object (handle)	36
			37
C binding			38
int MPI_F		fh, void *buf, int count,	39
	MPI_Datatype datatype	e, MPI_Request *request)	40 41
	008 binding		42
		nt, datatype, request, ierror)	43
	MPI_File), INTENT(IN) :: *), DIMENSION(), ASYNCH		44
	ER, INTENT(IN) :: count	ntonood bul	45
	MPI_Datatype), INTENT(IN)	:: datatype	$\frac{46}{47}$
	MPI_Request), INTENT(OUT)		48

```
626
```

```
1
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
     Fortran binding
3
     MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
4
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
5
          <type> BUF(*)
6
7
          MPI_FILE_IREAD_SHARED is a nonblocking version of the MPI_FILE_READ_SHARED
8
     interface.
9
10
     MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)
11
12
       INOUT
                 fh
                                              file handle (handle)
13
       IN
                 buf
                                              initial address of buffer (choice)
14
       IN
                 count
                                              number of elements in buffer (integer)
15
16
                                              datatype of each buffer element (handle)
       IN
                 datatype
17
       OUT
                                              request object (handle)
                 request
18
19
     C binding
20
     int MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count,
21
                     MPI_Datatype datatype, MPI_Request *request)
22
23
     Fortran 2008 binding
^{24}
     MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
25
          TYPE(MPI_File), INTENT(IN) :: fh
26
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
27
          INTEGER, INTENT(IN) :: count
28
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
          TYPE(MPI_Request), INTENT(OUT) :: request
30
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
     Fortran binding
32
     MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
33
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
34
          <type> BUF(*)
35
36
          MPI_FILE_IWRITE_SHARED is a nonblocking version of the
37
     MPI_FILE_WRITE_SHARED interface.
38
39
     Collective Operations
40
     The semantics of a collective access using a shared file pointer is that the accesses to the
41
42
     file will be in the order determined by the ranks of the processes within the group. For each
     process, the location in the file at which data is accessed is the position at which the shared
43
     file pointer would be after all processes whose ranks within the group less than that of this
44
     process had accessed their data. In addition, in order to prevent subsequent shared offset
45
     accesses by the same processes from interfering with this collective access, the call might
46
47
     return only after all the processes within the group have initiated their accesses. When the
48
```

call returns, the shared file pointer points to the next etype accessible, according to the file view used by all processes, after the last etype requested.

Advice to users. There may be some programs in which all processes in the group need to access the file using the shared file pointer, but the program may not *require* that data be accessed in order of process rank. In such programs, using the shared ordered routines (e.g., MPI_FILE_WRITE_ORDERED rather than MPI_FILE_WRITE_SHARED) may enable an implementation to optimize access, improving performance. (*End of advice to users.*)

Advice to implementors. Accesses to the data requested by all processes do not have to be serialized. Once all processes have issued their requests, locations within the file for all accesses can be computed, and accesses can proceed independently from each other, possibly in parallel. (*End of advice to implementors.*)

MPI_FILE_READ_ORDERED(fh, buf, count, datatype, status)

		uni, datatype, statusj	18
INOUT	fh	file handle (handle)	19
OUT	buf	initial address of buffer (choice)	20
IN	count	number of elements in buffer (integer)	21
IN	datatype	datatype of each buffer element (handle)	22
	51		23 24
OUT	status	status object (Status)	24 25
			26
C binding		fly and the first sound	27
int MPI_F		e fh, void *buf, int count, e, MPI_Status *status)	28
	MPI_Datatype datatyp	e, mri_Status *status)	29
	2008 binding		30
		mt, datatype, status, ierror)	31
	<pre>MPI_File), INTENT(IN) ::</pre>	fh	32
	(*), DIMENSION() :: buf		33
	ER, INTENT(IN) :: count		34
	[MPI_Datatype), INTENT(IN) [MPI_Status) :: status) :: datatype	35
	ER, OPTIONAL, INTENT(OUT)) iorror	36
	ER, OFIIONAL, INIENI(001)		37 38
Fortran b	oinding		39
		JNT, DATATYPE, STATUS, IERROR)	40
		STATUS(MPI_STATUS_SIZE), IERROR	41
<type< td=""><td>> BUF(*)</td><td></td><td>42</td></type<>	> BUF(*)		42
MPI_F	FILE_READ_ORDERED is a co	ollective version of the MPI_FILE_READ_SHARED	43
interface.			44
			45
			46
			47

```
1
     MPI_FILE_WRITE_ORDERED(fh, buf, count, datatype, status)
2
       INOUT
                fh
                                            file handle (handle)
3
       IN
                 buf
                                            initial address of buffer (choice)
4
5
       IN
                count
                                            number of elements in buffer (integer)
6
       IN
                                            datatype of each buffer element (handle)
                datatype
7
       OUT
                status
                                            status object (Status)
8
9
     C binding
10
11
     int MPI_File_write_ordered(MPI_File fh, const void *buf, int count,
                    MPI_Datatype datatype, MPI_Status *status)
12
13
     Fortran 2008 binding
14
     MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
15
          TYPE(MPI_File), INTENT(IN) :: fh
16
          TYPE(*), DIMENSION(...), INTENT(IN) :: buf
17
          INTEGER, INTENT(IN) :: count
18
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
          TYPE(MPI_Status) :: status
20
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     Fortran binding
23
     MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
24
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
25
          <type> BUF(*)
26
          MPI_FILE_WRITE_ORDERED is a collective version of the MPI_FILE_WRITE_SHARED
27
     interface.
28
29
     Seek
30
^{31}
     If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous
32
     to call the following two routines (MPI_FILE_SEEK_SHARED and
33
     MPI_FILE_GET_POSITION_SHARED).
34
35
     MPI_FILE_SEEK_SHARED(fh, offset, whence)
36
37
       INOUT
                 fh
                                            file handle (handle)
38
       IN
                 offset
                                            file offset (integer)
39
       IN
                whence
                                            update mode (state)
40
41
42
     C binding
43
     int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)
44
     Fortran 2008 binding
45
     MPI_File_seek_shared(fh, offset, whence, ierror)
46
          TYPE(MPI_File), INTENT(IN) :: fh
47
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
48
```

INTEGER, INTENT(IN) :: whence INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 2
Fortran binding	3
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)	4
INTEGER FH, WHENCE, IERROR	5
INTEGER FR, WHENCE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	6
INTEGER(KIND-MFI_OFFSEI_KIND) OFFSEI	7
MPI_FILE_SEEK_SHARED updates the shared file pointer according to whence, which has the following possible values:	8 9
• MPI_SEEK_SET: the pointer is set to offset	10 11
• MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset	12 13
• MPI_SEEK_END: the pointer is set to the end of file plus offset	14
MPI_FILE_SEEK_SHARED is collective; all the processes in the communicator group	15 16
associated with the file handle fh must call MPI_FILE_SEEK_SHARED with the same values	
for offset and whence.	17
The offset can be negative, which allows seeking backwards. It is erroneous to seek to	
a negative position in the view.	20
	20
	22
MPI_FILE_GET_POSITION_SHARED(fh, offset)	23
IN fh file handle (handle)	24
OUT offset of shared pointer (integer)	25
onset of shared pointer (integer)	26
Chinding	27
C binding int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)	28
Int MF1_FILe_get_position_shared(MF1_FILe In, MF1_011Set *011Set)	29
Fortran 2008 binding	30
MPI_File_get_position_shared(fh, offset, ierror)	31
TYPE(MPI_File), INTENT(IN) :: fh	32
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset	33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
Fortran binding	35
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)	36
INTEGER FH, IERROR	37
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	38
	39
MPI_FILE_GET_POSITION_SHARED returns, in offset, the current position of the	40
shared file pointer in etype units relative to the current view.	
	41
	42
Advice to users. The offset can be used in a future call to MPI_FILE_SEEK_SHARED	42 43
<i>Advice to users.</i> The offset can be used in a future call to MPI_FILE_SEEK_SHARED using whence = MPI_SEEK_SET to return to the current position. To set the displace-	42 43 44
Advice to users. The offset can be used in a future call to MPI_FILE_SEEK_SHARED	42 43 44 45

the resulting displacement. (End of advice to users.)

1 14.4.5 Split Collective Data Access Routines 2 MPI provides a restricted form of "nonblocking collective" I/O operations for all data ac-3 cesses using split collective data access routines. These routines are referred to as "split" 4 collective routines because a single collective operation is split in two: a begin routine and 5an end routine. The begin routine begins the operation, much like a nonblocking data access 6 (e.g., MPI_FILE_IREAD). The end routine completes the operation, much like the matching 7 test or wait (e.g., MPI_WAIT). As with nonblocking data access operations, the user must 8 not use the buffer passed to a begin routine while the routine is outstanding; the operation 9 must be completed with an end routine before it is safe to free buffers, etc. 10 Split collective data access operations on a file handle fh are subject to the semantic 11 rules given below. 1213 • On any MPI process, each file handle may have at most one active split collective 14operation at any time. 1516• Begin calls are collective over the group of processes that participated in the collective 17 open and follow the ordering rules for collective calls. 18 • End calls are collective over the group of processes that participated in the collective 19open and follow the ordering rules for collective calls. Each end call matches the 20preceding begin call for the same collective operation. When an "end" call is made. 21exactly one unmatched "begin" call for the same operation must precede it. 2223• An implementation is free to implement any split collective data access routine using 24the corresponding blocking collective routine when either the begin call (e.g., 25MPI_FILE_READ_ALL_BEGIN) or the end call (e.g., MPI_FILE_READ_ALL_END) is 26issued. The begin and end calls are provided to allow the user and MPI implementation 27to optimize the collective operation. 28According to the definitions in Section 2.4.2, the begin procedures are incomplete. 29 They are also non-local procedures because they may or may not return before they 30 are called in all MPI processes of the process group. 3132 This is one of the exceptions in which incomplete procedures Advice to users. 33 are non-local and therefore blocking. (End of advice to users.) 34 35• Split collective operations do not match the corresponding regular collective opera-36 tion. For example, in a single collective read operation, an MPI_FILE_READ_ALL 37 on one process does not match an MPI_FILE_READ_ALL_BEGIN/ 38 MPI_FILE_READ_ALL_END pair on another process. 39 • Split collective routines must specify a buffer in both the begin and end routines. 40 By specifying the buffer that receives data in the end routine, we can avoid the 41 problems described in "A Problem with Code Movements and Register Optimization," 42Section 19.1.17, but not all of the problems, such as those described in Sections 19.1.12, 43 19.1.13, and 19.1.16. 4445• No collective I/O operations are permitted on a file handle concurrently with a split 46collective access on that file handle (i.e., between the begin and end of the access). 47 That is 48

1 MPI_File_read_all_begin(fh, ...); 2 . . . MPI_File_read_all(fh, ...); . . . MPI_File_read_all_end(fh, ...); 5 6 is erroneous. • In a multithreaded implementation, any split collective begin and end operation called 9 by a process must be called from the same thread. This restriction is made to simplify 10 the implementation in the multithreaded case. (Note that we have already disallowed 11 having two threads begin a split collective operation on the same file handle since only 12one split collective operation can be active on a file handle at any time.) 13 14The arguments for these routines have the same meaning as for the equivalent collective 15versions (e.g., the argument definitions for MPI_FILE_READ_ALL_BEGIN and 16MPI_FILE_READ_ALL_END are equivalent to the arguments for MPI_FILE_READ_ALL). 17 The begin routine (e.g., MPI_FILE_READ_ALL_BEGIN) begins a split collective operation 18 that, when completed with the matching end routine (i.e., MPI_FILE_READ_ALL_END) 19 produces the result as defined for the equivalent collective routine (i.e., 20MPI_FILE_READ_ALL). 21For the purpose of consistency semantics (Section 14.6.1), a matched pair of split 22 collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and 23MPI_FILE_READ_ALL_END) compose a single data access. 24 2526MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype) 27IN fh file handle (handle) 28 IN offset file offset (integer) 29 30 OUT initial address of buffer (choice) buf 31 IN count number of elements in buffer (integer) 32 IN datatype datatype of each buffer element (handle) 33 34 C binding 35int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf, 36 int count, MPI_Datatype datatype) 37 38 Fortran 2008 binding 39 MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror) 40 TYPE(MPI_File), INTENT(IN) :: fh 41 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 42TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 43 INTEGER, INTENT(IN) :: count 44 TYPE(MPI_Datatype), INTENT(IN) :: datatype 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4647Fortran binding 48 MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)

```
1
          INTEGER FH, COUNT, DATATYPE, IERROR
\mathbf{2}
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
3
          <type> BUF(*)
4
5
6
     MPI_FILE_READ_AT_ALL_END(fh, buf, status)
7
       IN
                 fh
                                             file handle (handle)
8
       OUT
                 buf
9
                                             initial address of buffer (choice)
10
       OUT
                 status
                                             status object (Status)
11
12
     C binding
13
     int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
14
15
     Fortran 2008 binding
16
     MPI_File_read_at_all_end(fh, buf, status, ierror)
17
          TYPE(MPI_File), INTENT(IN) :: fh
18
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
19
          TYPE(MPI_Status) :: status
20
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     Fortran binding
22
     MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
23
          INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
24
          <type> BUF(*)
25
26
27
     MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
28
29
       INOUT
                 fh
                                             file handle (handle)
30
       IN
                 offset
                                             file offset (integer)
^{31}
                 buf
                                             initial address of buffer (choice)
       IN
32
33
       IN
                 count
                                             number of elements in buffer (integer)
34
       IN
                 datatype
                                             datatype of each buffer element (handle)
35
36
     C binding
37
     int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset,
38
                    const void *buf, int count, MPI_Datatype datatype)
39
40
     Fortran 2008 binding
41
     MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
42
          TYPE(MPI_File), INTENT(IN) :: fh
43
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
44
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
45
          INTEGER, INTENT(IN) :: count
46
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
47
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

INTEG INTEG	0		1 2 3 4 5 6 7
MPI_FILE_	WRITE_AT_ALL_END(fh, but	f, status)	8
INOUT	fh	file handle (handle)	9 10
IN	buf	initial address of buffer (choice)	10
OUT	status	status object (Status)	12
C binding	5	File fh, const void *buf,	13 14 15 16 17
Fortran 2	008 binding		18
	write_at_all_end(fh, buf;	, status, ierror)	19
	<pre>MPI_File), INTENT(IN) ::</pre>		20
		I(IN), ASYNCHRONOUS :: buf	21 22
	MPI_Status) :: status ER, OPTIONAL, INTENT(OUT)	·· ierror	22
			24
Fortran b	WRITE_AT_ALL_END(FH, BUF,	STATUS LEBROR)	25
	ER FH, STATUS(MPI_STATUS)		26
	> BUF(*)		27 28
			29
			30
	.READ_ALL_BEGIN(fh, buf, co	ount, datatype)	31
INOUT	fh	file handle (handle)	32 33
OUT	buf	initial address of buffer (choice)	34
1N	count	number of elements in buffer (integer)	35
IN	datatype	datatype of each buffer element (handle)	36
			37
C binding			38 39
int MPI_F	J J J J J J J J J J J J J J J J J J J	lle fh, void *buf, int count,	40
	MPI_Datatype datatyp	e)	41
	008 binding		42
	read_all_begin(fh, buf, o		43
	<pre>MPI_File), INTENT(IN) :: *), DIMENSION(), ASYNCH</pre>		44 45
	ER, INTENT(IN) :: count	MUDROOD DUI	45 46
	MPI_Datatype), INTENT(IN)	:: datatype	47
	ER, OPTIONAL, INTENT(OUT)		48

```
1
     Fortran binding
\mathbf{2}
     MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
3
          INTEGER FH, COUNT, DATATYPE, IERROR
4
          <type> BUF(*)
5
6
\overline{7}
     MPI_FILE_READ_ALL_END(fh, buf, status)
8
       INOUT
                 fh
                                             file handle (handle)
9
10
       OUT
                 buf
                                             initial address of buffer (choice)
11
       OUT
                                             status object (Status)
                 status
12
13
     C binding
14
     int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)
15
16
     Fortran 2008 binding
17
     MPI_File_read_all_end(fh, buf, status, ierror)
18
          TYPE(MPI_File), INTENT(IN) :: fh
19
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
          TYPE(MPI_Status) :: status
20
21
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     Fortran binding
23
     MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
24
          INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
25
          <type> BUF(*)
26
27
28
     MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype)
29
30
       INOUT
                 fh
                                             file handle (handle)
31
       IN
                 buf
                                             initial address of buffer (choice)
32
       IN
                 count
                                             number of elements in buffer (integer)
33
34
       IN
                 datatype
                                             datatype of each buffer element (handle)
35
36
     C binding
37
     int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count,
38
                    MPI_Datatype datatype)
39
     Fortran 2008 binding
40
     MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
41
          TYPE(MPI_File), INTENT(IN) :: fh
42
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
43
          INTEGER, INTENT(IN) :: count
44
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
45
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     Fortran binding
48
```

14.4. DAT	TA ACCESS		635
INTEG	WRITE_ALL_BEGIN(FH, BUF, ER FH, COUNT, DATATYPE, I > BUF(*)	COUNT, DATATYPE, IERROR) IERROR	1 2 3 4 5
MPI_FILE_	WRITE_ALL_END(fh, buf, sta	atus)	6
INOUT	fh	file handle (handle)	7
IN	buf	initial address of buffer (choice)	9
OUT	status	status object (Status)	10
C binding	S		11 12 13
int MPI_F	ile_write_all_end(MPI_Fi MPI_Status *status)	ie in, const void *bui,	14 15
Fortran 2	008 binding		16
	write_all_end(fh, buf, s		17
	<pre>MPI_File), INTENT(IN) :: *) DIMENSION() INTEN'</pre>	fh Γ(IN), ASYNCHRONOUS :: buf	18 19
	MPI_Status) :: status	I(IN), ASINGRONOUS Dui	20
	ER, OPTIONAL, INTENT(OUT)) :: ierror	21
Fortran b	inding		22
	WRITE_ALL_END(FH, BUF, S'	TATUS, IERROR)	23
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR			24 25
<type< td=""><td>> BUF(*)</td><td></td><td>26</td></type<>	> BUF(*)		26
			27
			28
	READ_ORDERED_BEGIN(fh,	but, count, datatype)	29
INOUT	fh	file handle (handle)	30 31
OUT	buf	initial address of buffer (choice)	32
IN	count	number of elements in buffer (integer)	33
IN	datatype	datatype of each buffer element (handle)	34
			35
C binding	g		36 37
int MPI_F		PI_File fh, void *buf, int count,	38
	MPI_Datatype datatyp	e)	39
	008 binding		40
	-	uf, count, datatype, ierror)	41
	<pre>MPI_File), INTENT(IN) :: *), DIMENSION(), ASYNCI</pre>		42
	ER, INTENT(IN) :: count	mondob bui	43 44
	MPI_Datatype), INTENT(IN)) :: datatype	45
	ER, OPTIONAL, INTENT(OUT)		46
Fortran h	inding		47
Fortran binding		48	

```
1
     MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
\mathbf{2}
         INTEGER FH, COUNT, DATATYPE, IERROR
3
         <type> BUF(*)
4
5
6
     MPI_FILE_READ_ORDERED_END(fh, buf, status)
7
       INOUT
                fh
                                            file handle (handle)
8
       OUT
                 buf
9
                                            initial address of buffer (choice)
10
       OUT
                status
                                            status object (Status)
11
12
     C binding
13
     int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
14
15
     Fortran 2008 binding
16
     MPI_File_read_ordered_end(fh, buf, status, ierror)
17
         TYPE(MPI_File), INTENT(IN) :: fh
18
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
19
         TYPE(MPI_Status) :: status
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     Fortran binding
22
     MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
23
          INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
24
         <type> BUF(*)
25
26
27
     MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)
28
29
       INOUT
                fh
                                            file handle (handle)
30
       IN
                 buf
                                            initial address of buffer (choice)
^{31}
                                            number of elements in buffer (integer)
       IN
                 count
32
33
       IN
                 datatype
                                            datatype of each buffer element (handle)
34
35
     C binding
36
     int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
37
                    MPI_Datatype datatype)
38
     Fortran 2008 binding
39
     MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
40
         TYPE(MPI_File), INTENT(IN) :: fh
41
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
42
         INTEGER, INTENT(IN) :: count
43
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     Fortran binding
47
     MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
48
```

```
INTEGER FH, COUNT, DATATYPE, IERROR
                                                                                        2
    <type> BUF(*)
MPI_FILE_WRITE_ORDERED_END(fh, buf, status)
 INOUT
           fh
                                      file handle (handle)
 IN
           buf
                                      initial address of buffer (choice)
 OUT
           status
                                      status object (Status)
                                                                                       10
                                                                                       11
C binding
                                                                                       12
int MPI_File_write_ordered_end(MPI_File fh, const void *buf,
                                                                                       13
              MPI_Status *status)
                                                                                       14
Fortran 2008 binding
                                                                                       15
MPI_File_write_ordered_end(fh, buf, status, ierror)
                                                                                       16
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                       17
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                       18
    TYPE(MPI_Status) :: status
                                                                                       19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       20
                                                                                       21
Fortran binding
                                                                                       22
MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
                                                                                       23
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
                                                                                       24
    <type> BUF(*)
                                                                                       25
                                                                                       26
```

14.5 File Interoperability

At the most basic level, file interoperability is the ability to read the information previously written to a file — not just the bits of data, but the actual information the bits represent. MPI guarantees full interoperability within a single MPI environment, and supports increased interoperability outside that environment through the external data representation (Section 14.5.2) as well as the data conversion functions (Section 14.5.3).

Interoperability within a single MPI environment (which could be considered "operability") ensures that file data written by one MPI process can be read by any other MPI process, subject to the consistency constraints (see Section 14.6.1), provided that it would have been possible to start the two processes simultaneously and have them reside in a single MPI_COMM_WORLD. Furthermore, both processes must see the same data values at every absolute byte offset in the file for which data was written.

This single environment file interoperability implies that file data is accessible regardless of the number of processes.

There are three aspects to file interoperability:

• transferring the bits,	44
	45
• converting between different file structures, and	40
• converting between different machine representations.	47
• converting between unterent machine representations.	48

Unofficial Draft for Comment Only

27

28 29

30

 31

32

33

34

35

36

37

38

39

40

41

42

1 The first two aspects of file interoperability are beyond the scope of this standard, $\mathbf{2}$ as both are highly machine dependent. However, transferring the bits of a file into and 3 out of the MPI environment (e.g., by writing a file to tape) is required to be supported 4 by all MPI implementations. In particular, an implementation must specify how familiar $\mathbf{5}$ operations similar to POSIX cp, rm, and mv can be performed on the file. Furthermore, it 6 is expected that the facility provided maintains the correspondence between absolute byte $\overline{7}$ offsets (e.g., after possible file structure conversion, the data bits at byte offset 102 in the 8 MPI environment are at byte offset 102 outside the MPI environment). As an example, 9 a simple off-line conversion utility that transfers and converts files between the native file 10 system and the MPI environment would suffice, provided it maintained the offset coherence 11mentioned above. In a high-quality implementation of MPI, users will be able to manipulate 12MPI files using the same or similar tools that the native file system offers for manipulating 13 its files.

The remaining aspect of file interoperability, converting between different machine representations, is supported by the typing information specified in the etype and filetype. This facility allows the information in files to be shared between any two applications, regardless of whether they use MPI, and regardless of the machine architectures on which they run.

¹⁹ MPI supports multiple data representations: "native," "internal," and "external32." ²⁰ An implementation may support additional data representations. MPI also supports user-²¹ defined data representations (see Section 14.5.3). The "native" and "internal" data repre-²² sentations are implementation dependent, while the "external32" representation is common ²³ to all MPI implementations and facilitates file interoperability. The data representation is ²⁴ specified in the datarep argument to MPI_FILE_SET_VIEW.

Advice to users. MPI is not guaranteed to retain knowledge of what data representation was used when a file is written. Therefore, to correctly retrieve file data, an MPI application is responsible for specifying the same data representation as was used to create the file. (*End of advice to users.*)

"native" Data in this representation is stored in a file exactly as it is in memory. The advantage of this data representation is that data precision and I/O performance are not lost in type conversions with a purely homogeneous environment. The disadvantage is the loss of transparent interoperability within a heterogeneous MPI environment.

Advice to users. This data representation should only be used in a homogeneous MPI environment, or when the MPI application is capable of performing the data type conversions itself. (*End of advice to users.*)

Advice to implementors. When implementing read and write operations on top of MPI message-passing, the message data should be typed as MPI_BYTE to ensure that the message routines do not perform any type conversions on the data. (*End of advice to implementors.*)

"internal" This data representation can be used for I/O operations in a homogeneous or heterogeneous environment; the implementation will perform type conversions if necessary. The implementation is free to store data in any format of its choice, with the restriction that it will maintain constant extents for all predefined datatypes in any

25 26

27

28

29 30

 31

32

33

34 35

36

37

38 39

40

41

42

one file. The environment in which the resulting file can be reused is implementationdefined and must be documented by the implementation.

Rationale. This data representation allows the implementation to perform I/O efficiently in a heterogeneous environment, though with implementation-defined restrictions on how the file can be reused. (*End of rationale.*)

Advice to implementors. Since "external32" is a superset of the functionality provided by "internal," an implementation may choose to implement "internal" as "external32." (*End of advice to implementors.*)

"external32" This data representation states that read and write operations convert all data from and to the "external32" representation defined in Section 14.5.2. The data conversion rules for communication also apply to these conversions (see Section 3.3.2). The data on the storage medium is always in this canonical representation, and the data in memory is always in the local process's native representation.

This data representation has several advantages. First, all processes reading the file in a heterogeneous MPI environment will automatically have the data converted to their respective native representations. Second, the file can be exported from one MPI environment and imported into any other MPI environment with the guarantee that the second environment will be able to read all the data in the file.

The disadvantage of this data representation is that data precision and I/O performance may be lost in data type conversions.

Advice to implementors. When implementing read and write operations on top of MPI message-passing, the message data should be converted to and from the "external32" representation in the client, and sent as type MPI_BYTE. This will avoid possible double data type conversions and the associated further loss of precision and performance. (*End of advice to implementors.*)

14.5.1 Datatypes for File Interoperability

If the file data representation is other than "native," care must be taken in constructing etypes and filetypes. Any of the datatype constructor functions may be used; however, for those functions that accept displacements in bytes, the displacements must be specified in terms of their values in the file for the file data representation being used. MPI will interpret these byte displacements as is; no scaling will be done. The function MPI_FILE_GET_TYPE_EXTENT can be used to calculate the extents of datatypes in the file. For etypes and filetypes that are portable datatypes (see Section 2.4), MPI will scale any displacements in the datatypes to match the file data representation. Datatypes passed as arguments to read/write routines specify the data layout in memory; therefore, they must always be constructed using displacements corresponding to displacements in memory.

Advice to users. One can logically think of the file as if it were stored in the memory of a file server. The etype and filetype are interpreted as if they were defined at this file server, by the same sequence of calls used to define them at the calling process. If the data representation is "native", then this logical file server runs on the same architecture as the calling process, so that these types define the same data layout

Unofficial Draft for Comment Only

 $\mathbf{2}$

1 on the file as they would define in the memory of the calling process. If the etype 2 and filetype are portable datatypes, then the data layout defined in the file is the 3 same as would be defined in the calling process memory, up to a scaling factor. The 4 routine MPI_FILE_GET_TYPE_EXTENT can be used to calculate this scaling factor. 5Thus, two equivalent, portable datatypes will define the same data layout in the file, 6 even in a heterogeneous environment with "internal", "external32", or user defined 7 data representations. Otherwise, the etype and filetype must be constructed so that 8 their typemap and extent are the same on any architecture. This can be achieved 9 if they have an explicit upper bound and lower bound (defined using

10 MPI_TYPE_CREATE_RESIZED). This condition must also be fulfilled by any datatype that is used in the construction of the etype and filetype, if this datatype is replicated 12contiguously, either explicitly, by a call to MPI_TYPE_CONTIGUOUS, or implicitly, 13 by a blocklength argument that is greater than one. If an etype or filetype is not 14portable, and has a typemap or extent that is architecture dependent, then the data 15layout specified by it on a file is implementation dependent. 16

File data representations other than "native" may be different from corresponding data representations in memory. Therefore, for these file data representations, it is important not to use hardwired byte offsets for file positioning, including the initial displacement that specifies the view. When a portable datatype (see Section 2.4) is used in a data access operation, any holes in the datatype are scaled to match the data representation. However, note that this technique only works when all the processes that created the file view build their etypes from the same predefined datatypes. For example, if one process uses an etype built from MPI_INT and another uses an etype built from MPI_FLOAT, the resulting views may be nonportable because the relative sizes of these types may differ from one data representation to another. (End of advice to users.)

```
28
29
     MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent)
30
^{31}
       IN
                fh
                                            file handle (handle)
32
       IN
                datatype
                                            datatype (handle)
33
       OUT
                extent
                                            datatype extent (integer)
34
35
36
     C binding
37
     int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
38
                    MPI_Aint *extent)
39
     Fortran 2008 binding
40
     MPI_File_get_type_extent(fh, datatype, extent, ierror)
41
         TYPE(MPI_File), INTENT(IN) :: fh
42
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     Fortran binding
47
     MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)
48
         INTEGER FH, DATATYPE, IERROR
                            Unofficial Draft for Comment Only
```

11

17

18

19

20

21

22

23

24

25

26

INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT

Returns the extent of datatype in the file fh. This extent will be the same for all processes accessing the file fh. If the current view uses a user-defined data representation (see Section 14.5.3), MPI uses the dtype_file_extent_fn callback to calculate the extent.

Advice to implementors. In the case of user-defined data representations, the extent of a derived datatype can be calculated by first determining the extents of the predefined datatypes in this derived datatype using dtype_file_extent_fn (see Section 14.5.3). (End of advice to implementors.)

14.5.2 External Data Representation: "external32"

All MPI implementations are required to support the data representation defined in this section. Support of optional datatypes (e.g., MPI_INTEGER2) is not required.

All floating point values are in big-endian IEEE format [42] of the appropriate size. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single (binary32)," "Double (binary64)," and "Double Extended (binary128)" formats, requiring 4, 8, and 16 bytes of storage, respectively. For the IEEE "Double Extended 18(binary128)" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, 1920bias = +16383, 112 fraction bits, and an encoding analogous to the "Double (binary64)" 21format. All integral values are in two's complement big-endian format. Big-endian means most significant byte at lowest address byte. For C _Bool, Fortran LOGICAL, and C++ bool, 220 implies false and nonzero implies true. C float _Complex, double _Complex, and long 23 24 double _Complex, Fortran COMPLEX and DOUBLE COMPLEX, and other complex types are represented by a pair of floating point format values for the real and imaginary components. Characters are in ISO 8859-1 format [43]. Wide characters (of type MPI_WCHAR) are in 26Unicode format [68]. 27

All signed numerals (e.g., MPI_INT, MPI_REAL) have the sign bit at the most significant bit. MPI_COMPLEX and MPI_DOUBLE_COMPLEX have the sign bit of the real and imaginary parts at the most significant bit of each part.

According to IEEE specifications [42], the "NaN" (not a number) is system dependent. It should not be interpreted within MPI as anything other than "NaN."

Advice to implementors. The MPI treatment of "NaN" is similar to the approach used in XDR [65]. (End of advice to implementors.)

All data is byte aligned, regardless of type. All data items are stored contiguously in the file (if the file view is contiguous).

Advice to implementors. All bytes of LOGICAL and bool must be checked to determine the value. (End of advice to implementors.)

Advice to users. The type MPI_PACKED is treated as bytes and is not converted. The user should be aware that MPI_PACK has the option of placing a header in the beginning of the pack buffer. (End of advice to users.)

The sizes of the predefined datatypes returned from MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_COMPLEX, and MPI_TYPE_CREATE_F90_INTEGER are defined in Section 19.1.9, page 753.

Unofficial Draft for Comment Only

1 $\mathbf{2}$

3

4

56

 $\overline{7}$

8

9

1213

1415

16

17

25

28

2930

31

32 33

34

3536

37

38 39

40

41 42

43

44

47

48

10 11

1	Predefined Type	Length	
2	MPI_PACKED	1	
3	MPI_BYTE	1	
4	MPI_CHAR	1	
5	MPI_UNSIGNED_CHAR	1	
6	MPI_SIGNED_CHAR	1	
7	MPI_SIGNED_CHAR	$\frac{1}{2}$	
8	MPI_SHORT	$\frac{2}{2}$	
9	MPI_SHORT MPI_UNSIGNED_SHORT	$\frac{2}{2}$	
10	MPI_UNSIGNED_SHORT	4	
11	MPI_LONG	4	
12			
13		4	
14	MPI_UNSIGNED_LONG	4	
14	MPI_LONG_LONG_INT	8	
16	MPI_UNSIGNED_LONG_LONG	8	
17	MPI_FLOAT	4	
18		8	
19	MPI_LONG_DOUBLE	16	
20	MPI_C_BOOL	1	
20	MPI_INT8_T	1	
22	MPI_INT16_T	2	
23	MPI_INT32_T	4	
24	MPI_INT64_T	8	
25	MPI_UINT8_T	1	
26	MPI_UINT16_T	2	
27	MPI_UINT32_T	4	
28	MPI_UINT64_T	8	
28	MPI_AINT	8	
30	MPI_COUNT	8	
31	MPI_OFFSET	$\frac{8}{2^*4}$	
32	MPI_C_COMPLEX	$2^{+}4$ $2^{*}4$	
33	MPI_C_FLOAT_COMPLEX		
34	MPI_C_DOUBLE_COMPLEX	2*8 2*16	
35	MPI_C_LONG_DOUBLE_COMPLEX		
36		1	
37		4	
38	MPI_INTEGER	4	
39	MPI_REAL	4	
40		8	
40		2*4	
41 42		2*8	
	MPI_CXX_BOOL	1	
43 44	MPI_CXX_FLOAT_COMPLEX	2*4	
44	MPI_CXX_DOUBLE_COMPLEX	2*8	
45 46	MPI_CXX_LONG_DOUBLE_COMPLEX	2*16	
40			

Table 14.2: "external32" sizes of predefined datatypes

Unofficial Draft for Comment Only

47

 48

Predefined Type	Length
MPI_INTEGER1	1
MPI_INTEGER2	2
MPI_INTEGER4	4
MPI_INTEGER8	8
MPI_INTEGER16	16
MPI_REAL2	2
MPI_REAL4	4
MPI_REAL8	8
MPI_REAL16	16
MPI_COMPLEX4	2*2
MPI_COMPLEX8	2*4
MPI_COMPLEX16	2*8
MPI_COMPLEX32	2*16

Table 14.3: "external32" sizes of optional datatypes

C++ Types	Length
MPI_CXX_BOOL	1
MPI_CXX_FLOAT_COMPLEX	$2^{*}4$
MPI_CXX_DOUBLE_COMPLEX	$2^{*}8$
MPI_CXX_LONG_DOUBLE_COMPLEX	2*16

\mathbf{pes}

Unofficial Draft for Comment Only

1 2 3 4 5	ir tl	teger, only the least significance of the sign bit value. This allows	then converting a larger size integer to a smaller size int bytes are moved. Care must be taken to preserve no conversion errors if the data range is within the r. (<i>End of advice to implementors.</i>)	
6 7 8	Table 14.2, 14.3, and 14.4 specify the sizes of predefined, optional, and C++ datatypes in "external32" format, respectively.			
9	14.5.3 User-Defined Data Representations			
10 11	There are two situations that cannot be handled by the required representations:			
12 13	1. a user wants to write a file in a representation unknown to the implementation, and			
14	2. a	user wants to read a file writte	n in a representation unknown to the implementation.	
15 16 17	16 User-defined data representations allow the user to insert a third party converter into the I/O stream to do the data representation conversion.			
18				
19 20	MFT_REGISTER_DATAREF (datarep, read_conversion_m, write_conversion_m,			
21 22	IN	datarep	data representation identifier (string)	
23 24	IN	read_conversion_fn	function invoked to convert from file representation to native representation (function)	
25 26	IN	write_conversion_fn	function invoked to convert from native representation to file representation (function)	
27 28 29	IN	dtype_file_extent_fn	function invoked to get the extent of a datatype as represented in the file (function)	
29 30	IN	extra_state	extra state	
31				
32	C bind	3		
33 34	int MP	I_Register_datarep(const	-	
35			<pre>rsion_function *read_conversion_fn, rsion_function *write_conversion_fn,</pre>	
36		-	function *dtype_file_extent_fn,	
37		void *extra_state)	· ·	
38	F (
39		n 2008 binding		
40	MP1_Re	• • • •	<pre>ead_conversion_fn, write_conversion_fn, _fn, extra_state, ierror)</pre>	
41	СН	ARACTER(LEN=*), INTENT(IN		
42			rsion_function) :: read_conversion_fn,	
43	110	write_conversion_		
44	PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn			
45	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state			
46 47		TEGER, OPTIONAL, INTENT(O		

⁴⁸ Fortran binding

```
MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
CHARACTER*(*) DATAREP
EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
INTEGER IERROR
```

The call associates read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn with the data representation identifier datarep. datarep can then be used as an argument to MPI_FILE_SET_VIEW, causing subsequent data access operations to call the conversion functions to convert all data items accessed between file data representation and native representation. MPI_REGISTER_DATAREP is a local operation and only registers the data representation for the calling MPI process. If datarep is already defined, an error in the error class MPI_ERR_DUP_DATAREP is raised using the default file error handler (see Section 14.7). The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING. MPI_MAX_DATAREP_STRING must have a value of at least 64. No routines are provided to delete data representations and free the associated resources; it is not expected that an application will generate them in significant numbers.

Extent Callback

The function dtype_file_extent_fn must return, in file_extent, the number of bytes required to store datatype in the file representation. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call. MPI will only call this routine with predefined datatypes employed by the user.

```
Datarep Conversion Functions
                                                                                    39
                                                                                    40
typedef int MPI_Datarep_conversion_function(void *userbuf,
                                                                                    41
              MPI_Datatype datatype, int count, void *filebuf,
                                                                                    42
              MPI_Offset position, void *extra_state);
                                                                                    43
ABSTRACT INTERFACE
                                                                                    44
  SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
                                                                                    45
              filebuf, position, extra_state, ierror)
                                                                                    46
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                    47
    TYPE(C_PTR), VALUE :: userbuf, filebuf
                                                                                    48
```

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

1	TYDE (MDI Detetyme) detetyme
2	TYPE(MPI_Datatype) :: datatype INTEGER :: count, ierror
3	INTEGER(KIND=MPI_OFFSET_KIND) :: position
4	INTEGER(KIND-MPI_OFFSEI_KIND) :: position INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
5	INTEGER(KIND=MP1_ADDRESS_KIND) :: extra_state
6	SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
7	POSITION, EXTRA_STATE, IERROR)
8	<type> USERBUF(*), FILEBUF(*)</type>
9	INTEGER DATATYPE, COUNT, IERROR
9 10	INTEGER(KIND=MPI_OFFSET_KIND) POSITION
11	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
12	
12	The function read_conversion_fn must convert from file data representation to na-
13	tive representation. Before calling this routine, MPI allocates and fills filebuf with count
14	contiguous data items. The type of each data item matches the corresponding entry for the
16	predefined datatype in the type signature of datatype. The function is passed, in extra_state,
17	the argument that was passed to the MPI_REGISTER_DATAREP call. The function must
18	copy all count data items from filebuf to userbuf in the distribution described by datatype,
19	converting each data item from file representation to native representation. datatype will be
20	equivalent to the datatype that the user passed to the read function. If the size of datatype
20	is less than the size of the count data items, the conversion function must treat datatype
21	as being contiguously tiled over the userbuf. The conversion function must begin storing
22	converted data at the location in userbuf specified by position into the (tiled) datatype.
23	Advice to users. Although the conversion functions have similarities to MPI_PACK
25	and MPI_UNPACK, one should note the differences in the use of the arguments count
26	and position. In the conversion functions, count is a count of data items (i.e., count
27	of typemap entries of datatype), and position is an index into this typemap. In
28	MPI_PACK, incount refers to the number of whole datatypes, and position is a number
29	of bytes. (End of advice to users.)
30	
31	Advice to implementors. A converted read operation could be implemented as follows:
32	
33	1. Get file extent of all data items
34	2. Allocate a filebuf large enough to hold all count data items
35	3. Read data from file into filebuf
36	
37	4. Call read_conversion_fn to convert data and place it into userbuf
38	5. Deallocate filebuf
39	(End of advice to implementance)
40	(End of advice to implementors.)
41	If MPI cannot allocate a buffer large enough to hold all the data to be converted from
42	a read operation, it may call the conversion function repeatedly using the same datatype
43	and userbuf, and reading successive chunks of data to be converted in filebuf. For the first
44	call (and in the case when all the data to be converted fits into filebuf), MPI will call the
45	function with position set to zero. Data converted during this call will be stored in the
46	userbuf according to the first count data items in datatype. Then in subsequent calls to the
47	conversion function, MPI will increment the value in position by the count of items converted
48	in the previous call, and the userbuf pointer will be unchanged.

Rationale. Passing the conversion function a position and one datatype for the transfer allows the conversion function to decode the datatype only once and cache an internal representation of it on the datatype. Then on subsequent calls, the conversion function can use the **position** to quickly find its place in the datatype and continue storing converted data where it left off at the end of the previous call. (*End of rationale.*)

Advice to users. Although the conversion function may usefully cache an internal representation on the datatype, it should not cache any state information specific to an ongoing conversion operation, since it is possible for the same datatype to be used concurrently in multiple conversion operations. (*End of advice to users.*)

The function write_conversion_fn must convert from native representation to file data representation. Before calling this routine, MPI allocates filebuf of a size large enough to hold count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function must copy count data items from userbuf in the distribution described by datatype, to a contiguous distribution in filebuf, converting each data item from native representation to file representation. If the size of datatype is less than the size of count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf.

The function must begin copying at the location in userbuf specified by position into the (tiled) datatype. datatype will be equivalent to the datatype that the user passed to the write function. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call.

The predefined constant MPI_CONVERSION_FN_NULL may be used as either write_conversion_fn or read_conversion_fn. In that case, MPI will not attempt to invoke write_conversion_fn or read_conversion_fn, respectively, but will perform the requested data access using the native data representation.

An MPI implementation must ensure that all data accessed is converted, either by using a filebuf large enough to hold all the requested data items or else by making repeated calls to the conversion function with the same datatype argument and appropriate values for position.

An implementation will only invoke the callback routines in this section (read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn) when one of the read or write routines in Section 14.4, or MPI_FILE_GET_TYPE_EXTENT is called by the user. dtype_file_extent_fn will only be passed predefined datatypes employed by the user. The conversion functions will only be passed datatypes equivalent to those that the user has passed to one of the routines noted above.

The conversion functions must be reentrant. User defined data representations are restricted to use byte alignment for all types. Furthermore, it is erroneous for the conversion functions to call any collective routines or to free datatype.

The conversion functions should return an error code. If the returned error code has a value other than MPI_SUCCESS, the implementation will raise an error in the class MPI_ERR_CONVERSION.

14.5.4 Matching Data Representations

It is the user's responsibility to ensure that the data representation used to read data from a file is *compatible* with the data representation that was used to write that data to the file.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\mathbf{5}$

 $\overline{7}$

 31

 $46 \\ 47$

In general, using the same data representation name when writing and reading a file does not guarantee that the representation is compatible. Similarly, using different representation names on two different implementations may yield compatible representations.

⁴ Compatibility can be obtained when "external32" representation is used, although
 ⁵ precision may be lost and the performance may be less than when "native" representation is
 ⁶ used. Compatibility is guaranteed using "external32" provided at least one of the following
 ⁷ conditions is met.

- The data access routines directly use types enumerated in Section 14.5.2, that are supported by all implementations participating in the I/O. The predefined type used to write a data item must also be used to read a data item.
- In the case of Fortran 90 programs, the programs participating in the data accesses obtain compatible datatypes using MPI routines that specify precision and/or range (Section 19.1.9).
- For any given data item, the programs participating in the data accesses use compatible predefined types to write and read the data item.

User-defined data representations may be used to provide an implementation compatibility with another implementation's "native" or "internal" representation.

Advice to users. Section 19.1.9 defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (End of advice to users.)

14.6 Consistency and Semantics

14.6.1 File Consistency

Consistency semantics define the outcome of multiple accesses to a single file. All file 30 accesses in MPI are relative to a specific file handle created from a collective open. MPI 31 32 provides three levels of consistency: sequential consistency among all accesses using a single file handle, sequential consistency among all accesses using file handles created from a single 33 34collective open with atomic mode enabled, and user-imposed consistency among accesses other than the above. Sequential consistency means the behavior of a set of operations will 35 be as if the operations were performed in some serial order consistent with program order; 36 each access appears atomic, although the exact ordering of accesses is unspecified. User-37 imposed consistency may be obtained using program order and calls to MPI_FILE_SYNC. 38

39 Let FH_1 be the set of file handles created from one particular collective open of the file FOO, and FH_2 be the set of file handles created from a different collective open of 40FOO. Note that nothing restrictive is said about FH_1 and FH_2 : the sizes of FH_1 and 41 FH_2 may be different, the groups of processes used for each open may or may not intersect, 42the file handles in FH_1 may be destroyed before those in FH_2 are created, etc. Consider 43the following three cases: a single file handle (e.g., $fh_1 \in FH_1$), two file handles created 44from a single collective open (e.g., $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$), and two file handles from 45different collective opens (e.g., $fh_1 \in FH_1$ and $fh_2 \in FH_2$). 46

⁴⁷ For the purpose of consistency semantics, a matched pair (Section 14.4.5) of split col-⁴⁸ lective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and

1

 $\mathbf{2}$

3

8 9

10

11 12

13

14

15 16

17

18 19

20

21

22

23

 24

25 26

27 28

MPI_FILE_READ_ALL_END) compose a single data access operation. Similarly, a nonblocking data access routine (e.g., MPI_FILE_IREAD) and the routine which completes the request (e.g., MPI_WAIT) also compose a single data access operation. For all cases below, these data access operations are subject to the same constraints as blocking data access operations.

Advice to users. For an MPI_FILE_IREAD and MPI_WAIT pair, the operation begins when MPI_FILE_IREAD is called and ends when MPI_WAIT returns. (*End of advice to users.*)

Assume that A_1 and A_2 are two data access operations. Let $D_1(D_2)$ be the set of absolute byte displacements of every byte accessed in $A_1(A_2)$. The two data accesses *overlap* if $D_1 \cap D_2 \neq \emptyset$. The two data accesses *conflict* if they overlap and at least one is a write access.

Let SEQ_{fh} be a sequence of file operations on a single file handle, bracketed by MPI_FILE_SYNCs on that file handle. (Both opening and closing a file implicitly perform an MPI_FILE_SYNC.) SEQ_{fh} is a "write sequence" if any of the data access operations in the sequence are writes or if any of the file manipulation operations in the sequence change the state of the file (e.g., MPI_FILE_SET_SIZE or MPI_FILE_PREALLOCATE). Given two sequences, SEQ_1 and SEQ_2 , we say they are not *concurrent* if one sequence is guaranteed to completely precede the other (temporally).

The requirements for guaranteeing sequential consistency among all accesses to a particular file are divided into the three cases given below. If any of these requirements are not met, then the value of all data in that file is implementation dependent.

Case 1: $fh_1 \in FH_1$ All operations on fh_1 are sequentially consistent if atomic mode is set. If nonatomic mode is set, then all operations on fh_1 are sequentially consistent if they are either nonconcurrent, nonconflicting, or both.

Case 2: $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$ Assume A_1 is a data access operation using fh_{1a} , and A_2 is a data access operation using fh_{1b} . If for any access A_1 , there is no access A_2 that conflicts with A_1 , then MPI guarantees sequential consistency.

However, unlike POSIX semantics, the default MPI semantics for conflicting accesses do not guarantee sequential consistency. If A_1 and A_2 conflict, sequential consistency can be guaranteed by either enabling atomic mode via the MPI_FILE_SET_ATOMICITY routine, or meeting the condition described in Case 3 below.

Case 3: $fh_1 \in FH_1$ and $fh_2 \in FH_2$ Consider access to a single file using file handles from distinct collective opens. In order to guarantee sequential consistency, MPI_FILE_SYNC must be used (both opening and closing a file implicitly perform an MPI_FILE_SYNC).

Sequential consistency is guaranteed among accesses to a single file if for any write sequence SEQ_1 to the file, there is no sequence SEQ_2 to the file which is *concurrent* with SEQ_1 . To guarantee sequential consistency when there are write sequences,

MPI_FILE_SYNC must be used together with a mechanism that guarantees nonconcurrency of the sequences.

See the examples in Section 14.6.11 for further clarification of some of these consistency semantics.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4 5

6

 $\overline{7}$

8

9 10

11

12

13

14

15

16

17

18

19

20

21

22

23

24 25

26

27

28 29

30 31

32 33

34

35 36

37 38

39

40

41

42

43

44

45

46

47

```
1
     MPI_FILE_SET_ATOMICITY(fh, flag)
2
       INOUT
                 fh
                                              file handle (handle)
3
                                              true to set atomic mode, false to set nonatomic mode
       IN
                 flag
4
                                              (logical)
5
6
7
     C binding
8
     int MPI_File_set_atomicity(MPI_File fh, int flag)
9
     Fortran 2008 binding
10
     MPI_File_set_atomicity(fh, flag, ierror)
11
          TYPE(MPI_File), INTENT(IN) :: fh
12
          LOGICAL, INTENT(IN) :: flag
13
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     Fortran binding
16
     MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
17
          INTEGER FH, IERROR
18
          LOGICAL FLAG
19
          Let FH be the set of file handles created by one collective open. The consistency
20
     semantics for data access operations using FH is set by collectively calling
21
     MPI_FILE_SET_ATOMICITY on FH. MPI_FILE_SET_ATOMICITY is collective; all pro-
22
     cesses in the group must pass identical values for fh and flag. If flag is true, atomic mode is
23
     set; if flag is false, nonatomic mode is set.
24
          Changing the consistency semantics for an open file only affects new data accesses.
25
     All completed data accesses are guaranteed to abide by the consistency semantics in effect
26
     during their execution. Nonblocking data accesses and split collective operations that have
27
     not completed (e.g., via MPI_WAIT) are only guaranteed to abide by nonatomic mode
28
     consistency semantics.
29
30
           Advice to implementors. Since the semantics guaranteed by atomic mode are stronger
31
           than those guaranteed by nonatomic mode, an implementation is free to adhere to
32
           the more stringent atomic mode semantics for outstanding requests. (End of advice
33
           to implementors.)
34
35
36
     MPI_FILE_GET_ATOMICITY(fh, flag)
37
38
       IN
                 fh
                                              file handle (handle)
39
       OUT
                 flag
                                              true if atomic mode, false if nonatomic mode (logical)
40
41
     C binding
42
     int MPI_File_get_atomicity(MPI_File fh, int *flag)
43
44
     Fortran 2008 binding
45
     MPI_File_get_atomicity(fh, flag, ierror)
46
          TYPE(MPI_File), INTENT(IN) :: fh
47
          LOGICAL, INTENT(OUT) :: flag
48
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2 Fortran binding 3 MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR 5 LOGICAL FLAG 6 7 MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access 8 operations on the set of file handles created by one collective open. If flag is true, atomic 9 mode is enabled; if flag is false, nonatomic mode is enabled. 10 11 MPI_FILE_SYNC(fh) 1213 INOUT fh file handle (handle) 1415C binding 16int MPI_File_sync(MPI_File fh) 17 Fortran 2008 binding 18 MPI_File_sync(fh, ierror) 19 TYPE(MPI_File), INTENT(IN) :: fh 20INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2122 Fortran binding 23MPI_FILE_SYNC(FH, IERROR) 24 INTEGER FH, IERROR 25Calling MPI_FILE_SYNC with fh causes all previous writes to fh by the calling process 26to be transferred to the storage device. If other processes have made updates to the storage 27device, then all such updates become visible to subsequent reads of fh by the calling process. 28MPI_FILE_SYNC may be necessary to ensure sequential consistency in certain cases (see 29above). 30

MPI_FILE_SYNC is a collective operation.

The user is responsible for ensuring that all nonblocking requests and split collective operations on fh have been completed before calling MPI_FILE_SYNC — otherwise, the call to MPI_FILE_SYNC is erroneous.

14.6.2 Random Access vs. Sequential Files

MPI distinguishes ordinary random access files from sequential stream files, such as pipes and tape files. Sequential stream files must be opened with the MPI_MODE_SEQUENTIAL flag set in the amode. For these files, the only permitted data access operations are shared file pointer reads and writes. Filetypes and etypes with holes are erroneous. In addition, the notion of file pointer is not meaningful; therefore, calls to MPI_FILE_SEEK_SHARED and MPI_FILE_GET_POSITION_SHARED are erroneous, and the pointer update rules specified for the data access routines do not apply. The amount of data accessed by a data access operation will be the amount requested unless the end of file is reached or an error is raised.

Rationale. This implies that reading on a pipe will always wait until the requested 46 amount of data is available or until the process writing to the pipe has issued an end of file. (End of rationale.) 47

Unofficial Draft for Comment Only

31

32

33

34 35

36 37

38

39

40

41

42

43

44

Finally, for some sequential files, such as those corresponding to magnetic tapes or streaming network connections, writes to the file may be destructive. In other words, a write may act as a truncate (a MPI_FILE_SET_SIZE with size set to the current position) followed by the write.

14.6.3 Progress

The progress rules of MPI are both a promise to users and a set of constraints on implementors. In cases where the progress rules restrict possible implementation choices more than the interface specification alone, the progress rules take precedence.

All blocking routines must complete in finite time unless an exceptional condition (such as resource exhaustion) causes an error.

Nonblocking data access routines inherit the following progress rule from nonblocking point to point communication: a nonblocking write is equivalent to a nonblocking send for which a receive is eventually posted, and a nonblocking read is equivalent to a nonblocking receive for which a send is eventually posted.

Finally, an implementation is free to delay progress of collective routines until all processes in the group associated with the collective call have invoked the routine. Once all processes in the group have invoked the routine, the progress rule of the equivalent noncollective routine must be followed.

14.6.4 Collective File Operations

²³ Collective file operations are subject to the same restrictions as collective communication ²⁴ operations. For a complete discussion, please refer to the semantics set forth in Section 6.14.

Collective file operations are collective over a duplicate of the communicator used to open the file — this duplicate communicator is implicitly specified via the file handle argument. Different processes can pass different values for other arguments of a collective routine unless specified otherwise.

29 30 31

21

22

26

27

28

14.6.5 Nonblocking Collective File Operations

³² Nonblocking collective file operations are defined only for data access routines with explicit
 ³³ offsets and individual file pointers but not with shared file pointers.

Nonblocking collective file operations are subject to the same restrictions as blocking collective I/O operations. All processes belonging to the group of the communicator that was used to open the file must call collective I/O operations (blocking and nonblocking) in the same order. This is consistent with the ordering rules for collective operations in threaded environments. For a complete discussion, please refer to the semantics set forth in Section 6.14.

⁴⁰ Nonblocking collective I/O operations do not match with blocking collective I/O oper ⁴¹ ations. Multiple nonblocking collective I/O operations can be outstanding on a single file
 ⁴² handle. High quality MPI implementations should be able to support a large number of
 ⁴³ pending nonblocking I/O operations.

⁴⁴ All nonblocking collective I/O calls are local and return immediately, irrespective of the ⁴⁵ status of other processes. The call initiates the operation which may progress independently ⁴⁶ of any communication, computation, or I/O. The call returns a request handle, which must ⁴⁷ be passed to a completion call. Input buffers should not be modified and output buffers ⁴⁸ should not be accessed before the completion call returns. The same progress rules described

Unofficial Draft for Comment Only

5 6

for nonblocking collective operations apply for nonblocking collective I/O operations. For a complete discussion, please refer to the semantics set forth in Section 6.12.

14.6.6 Type Matching

The type matching rules for I/O mimic the type matching rules for communication with one exception: if etype is MPI_BYTE, then this matches any datatype in a data access operation. In general, the etype of data items written must match the etype used to read the items, and for each data access operation, the current etype must also match the type declaration of the data access buffer.

Advice to users. In most cases, use of MPI_BYTE as a wild card will defeat the file interoperability features of MPI. File interoperability can only perform automatic conversion between heterogeneous data representations when the exact datatypes accessed are explicitly specified. (*End of advice to users.*)

14.6.7 Miscellaneous Clarifications

Once an I/O routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the comm and info used in an MPI_FILE_OPEN, or the etype and filetype used in an MPI_FILE_SET_VIEW, can be freed without affecting access to the file. Note that for nonblocking routines and split collective operations, the operation must be completed before it is safe to reuse data buffers passed as arguments.

As in communication, datatypes must be committed before they can be used in file manipulation or data access operations. For example, the etype and filetype must be committed before calling MPI_FILE_SET_VIEW, and the datatype must be committed before calling MPI_FILE_READ or MPI_FILE_WRITE.

14.6.8 MPI_Offset Type

MPI_Offset is an integer type of size sufficient to represent the size (in bytes) of the largest file supported by MPI. Displacements and offsets are always specified as values of type MPI_Offset.

In Fortran, the corresponding integer is an integer with kind parameter MPI_OFFSET_KIND, which is defined in the mpi_f08 module, the mpi module and the mpif.h include file.

In Fortran 77 environments that do not support KIND parameters, MPI_Offset arguments should be declared as an INTEGER of suitable size. The language interoperability implications for MPI_Offset are similar to those for addresses (see Section 19.2).

14.6.9 Logical vs. Physical File Layout

MPI specifies how the data should be laid out in a virtual file structure (the view), not how that file structure is to be stored on one or more disks. Specification of the physical file structure was avoided because it is expected that the mapping of files to disks will be system specific, and any specific control over file layout would therefore restrict program portability. However, there are still cases where some information may be necessary to optimize file layout. This information can be provided as *hints* specified via info when a file is created (see Section 14.2.8).

Unofficial Draft for Comment Only

 $\overline{7}$

 24

 31

1	14.6.10 File Size
2 3 4 5 6	The size of a file may be increased by writing to the file after the current end of file. The size may also be changed by calling MPI <i>size changing</i> routines, such as MPI_FILE_SET_SIZE. A call to a size changing routine does not necessarily change the file size. For example, calling MPI_FILE_PREALLOCATE with a size less than the current size does not change the size.
7 8 9 10	Consider a set of bytes that has been written to a file since the most recent call to a size changing routine, or since MPI_FILE_OPEN if no such routine has been called. Let the high byte be the byte in that set with the largest displacement. The file size is the larger of
11 12	• One plus the displacement of the high byte.
13	\bullet The size immediately after the size changing routine, or $MPI_FILE_OPEN,$ returned.
14 15 16 17 18 19	When applying consistency semantics, calls to MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE are considered writes to the file (which conflict with operations that access bytes at displacements between the old and new file sizes), and MPI_FILE_GET_SIZE is considered a read of the file (which overlaps with all accesses to the file).
20 21 22 23 24	Advice to users. Any sequence of operations containing the collective routines MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE is a write sequence. As such, sequential consistency in nonatomic mode is not guaranteed unless the conditions in Section 14.6.1 are satisfied. (<i>End of advice to users.</i>)
25 26 27	File pointer update semantics (i.e., file pointers are updated by the amount accessed) are only guaranteed if file size changes are sequentially consistent.
28 29 30 31 32 33 34	Advice to users. Consider the following example. Given two operations made by separate processes to a file containing 100 bytes: an MPI_FILE_READ of 10 bytes and an MPI_FILE_SET_SIZE to 0 bytes. If the user does not enforce sequential consistency between these two operations, the file pointer may be updated by the amount requested (10 bytes) even if the amount accessed is zero bytes. (<i>End of advice to users.</i>)
35	14.6.11 Examples
36 37 38	The examples in this section illustrate the application of the MPI consistency and semantics guarantees. These address
39 40	• conflicting accesses on file handles obtained from a single collective open, and
41	• all accesses on file handles obtained from two separate collective opens.
42 43 44 45 46	The simplest way to achieve consistency for conflicting accesses is to obtain sequential consistency by setting atomic mode. For the code below, process 1 will read either 0 or 10 integers. If the latter, every element of b will be 5 . If nonatomic mode is set, the results of the read are undefined.

```
1
/* Process 0 */
                                                                                        \mathbf{2}
int i, a[10];
                                                                                        3
int TRUE = 1;
                                                                                        4
for (i=0;i<10;i++)
                                                                                        5
                                                                                        6
   a[i] = 5;
                                                                                        7
MPI_File_open(MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
                                                                                        9
                                                                                        10
MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
                                                                                        11
MPI_File_set_atomicity(fh0, TRUE);
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
                                                                                        12
                                                                                        13
/* MPI_Barrier(MPI_COMM_WORLD); */
                                                                                        14
/* Process 1 */
                                                                                        15
int b[10];
                                                                                        16
int TRUE = 1;
                                                                                        17
MPI_File_open(MPI_COMM_WORLD, "workfile",
                                                                                        18
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
                                                                                        19
MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
                                                                                        20
MPI_File_set_atomicity(fh1, TRUE);
                                                                                        21
/* MPI_Barrier(MPI_COMM_WORLD); */
                                                                                        22
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
                                                                                        23
                                                                                        ^{24}
A user may guarantee that the write on process 0 precedes the read on process 1 by imposing
                                                                                        25
temporal order with, for example, calls to MPI_BARRIER.
                                                                                        26
                                                                                        27
     Advice to users. Routines other than MPI_BARRIER may be used to impose temporal
                                                                                        28
     order. In the example above, process 0 could use MPI_SEND to send a 0 byte message,
                                                                                        29
     received by process 1 using MPI_RECV. (End of advice to users.)
                                                                                        30
                                                                                        31
    Alternatively, a user can impose consistency with nonatomic mode set:
                                                                                        32
                                                                                        33
/* Process 0 */
                                                                                        34
int i, a[10];
                                                                                        35
for (i=0;i<10;i++)</pre>
                                                                                        36
   a[i] = 5;
                                                                                        37
                                                                                        38
MPI_File_open(MPI_COMM_WORLD, "workfile",
                                                                                        39
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
                                                                                        40
MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
                                                                                        41
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status );
                                                                                        42
MPI_File_sync(fh0);
                                                                                        43
MPI_Barrier(MPI_COMM_WORLD);
                                                                                        44
MPI_File_sync(fh0);
                                                                                        45
                                                                                        46
/* Process 1 */
                                                                                        47
int b[10];
                                                                                        48
MPI_File_open(MPI_COMM_WORLD, "workfile",
```

1	<pre>MPI_MODE_RDWR MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);</pre>
2	<pre>MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);</pre>
3	<pre>MPI_File_sync(fh1);</pre>
4	<pre>MPI_Barrier(MPI_COMM_WORLD);</pre>
5	<pre>MPI_File_sync(fh1);</pre>
6 7	<pre>MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);</pre>
8	The "sync-barrier-sync" construct is required because:
9 10	• The barrier ensures that the write on process 0 occurs before the read on process 1.
11 12	• The first sync guarantees that the data written by all processes is transferred to the storage device.
13 14 15	• The second sync guarantees that all data which has been transferred to the storage device is visible to all processes. (This does not affect process 0 in this example.)
16 17	The following program represents an erroneous attempt to achieve consistency by elim- inating the apparently superfluous second "sync" call for each process.
18	/* THIS EXAMPLE IS ERRONEOUS */
19	/* Process 0 */
20	int i, a[10];
21	for (i=0;i<10;i++)
22	a[i] = 5;
23	
24	MPI_File_open(MPI_COMM_WORLD, "workfile",
25 26	<pre>MPI_MODE_RDWR MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);</pre>
20	<pre>MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INF0_NULL);</pre>
28	<pre>MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);</pre>
29	<pre>MPI_File_sync(fh0);</pre>
30	<pre>MPI_Barrier(MPI_COMM_WORLD);</pre>
31	/* Process 1 */
32	int b[10];
33	MPI_File_open(MPI_COMM_WORLD, "workfile",
34	<pre>MPI_MODE_RDWR MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);</pre>
35	<pre>MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);</pre>
36	<pre>MPI_Barrier(MPI_COMM_WORLD);</pre>
37	<pre>MPI_File_sync(fh1);</pre>
38	<pre>MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);</pre>
39 40	/* THIS EXAMPLE IS ERRONEOUS */
41	
42	The above program also violates the MPI rule against out-of-order collective operations and
43	will deadlock for implementations in which MPI_FILE_SYNC blocks.
44	Advice to users. Some implementations may choose to implement MPI_FILE_SYNC
45	as a temporally synchronizing function. When using such an implementation, the
46	"sync-barrier-sync" construct above can be replaced by a single "sync." The results of
47	using such code with an implementation for which MPI_FILE_SYNC is not temporally
48	synchronizing is undefined. (End of advice to users.)

Asynchronous I/O

The behavior of asynchronous I/O operations is determined by applying the rules specified above for synchronous I/O operations.

The following examples all access a preexisting file "myfile." Word 10 in myfile initially contains the integer 2. Each example writes and reads word 10.

First consider the following code fragment:

For asynchronous data access operations, MPI specifies that the access occurs at any time between the call to the asynchronous data access routine and the return from the corresponding request complete routine. Thus, executing either the read before the write, or the write before the read is consistent with program order. If atomic mode is set, then MPI guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic mode is not set, then sequential consistency is not guaranteed and the program may read something other than 2 or 4 due to the conflicting data access.

Similarly, the following code fragment does not order file accesses:

```
int a = 4, b;
MPI_File_open(MPI_COMM_WORLD, "myfile",
              MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
/* MPI_File_set_atomicity(fh, TRUE); Use this to set atomic mode. */
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
MPI_Wait(&reqs[0], &status);
MPI_Wait(&reqs[1], &status);
If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee
sequential consistency in nonatomic mode.
    On the other hand, the following code fragment:
int a = 4, b;
MPI_File_open(MPI_COMM_WORLD, "myfile",
              MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
```

MPI_Wait(&reqs[0], &status); MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);

MPI_Wait(&reqs[1], &status);

defines the same ordering as:

2

3

4

5

6

7

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24 25

26

27

28

29

30

31

32

33

34 35

36

37

38

39

40

41

42

43

44

45

46 47

```
1
      int a = 4, b;
\mathbf{2}
     MPI_File_open(MPI_COMM_WORLD, "myfile",
3
                      MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
4
     MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
\mathbf{5}
     MPI_File_write_at(fh, 10, &a, 1, MPI_INT, &status );
6
     MPI_File_read_at(fh, 10, &b, 1, MPI_INT, &status );
7
      Since
8
9
         • nonconcurrent operations on a single file handle are sequentially consistent, and
10
11
         • the program fragments specify an order for the operations.
12
      MPI guarantees that both program fragments will read the value 4 into b. There is no need
13
      to set atomic mode for this example.
14
          Similar considerations apply to conflicting accesses of the form:
15
16
     MPI_File_iwrite_all(fh,...);
17
     MPI_File_iread_all(fh,...);
18
     MPI_Waitall(...);
19
          In addition, as mentioned in Section 14.6.5, nonblocking collective I/O operations have
20
      to be called in the same order on the file handle by all processes.
21
22
          Similar considerations apply to conflicting accesses of the form:
23
     MPI_File_write_all_begin(fh,...);
24
     MPI_File_iread(fh,...);
25
     MPI_Wait(fh,...);
26
     MPI_File_write_all_end(fh,...);
27
28
          Recall that constraints governing consistency and semantics are not relevant to the
29
     following:
30
^{31}
     MPI_File_write_all_begin(fh,...);
32
     MPI_File_read_all_begin(fh,...);
33
     MPI_File_read_all_end(fh,...);
34
     MPI_File_write_all_end(fh,...);
35
     since split collective operations on the same file handle may not overlap (see Section 14.4.5).
36
```

14.7 I/O Error Handling

37

38

39

44

⁴⁰ By default, communication errors are fatal — MPI_ERRORS_ARE_FATAL is the default error
⁴¹ handler associated with MPI_COMM_WORLD. I/O errors are usually less catastrophic (e.g.,
⁴² "file not found") than communication errors, and common practice is to catch these errors
⁴³ and continue executing. For this reason, MPI provides additional error facilities for I/O.

Advice to users. MPI does not specify the state of a computation after an erroneous
 MPI call has occurred. A high-quality implementation will support the I/O error
 handling facilities, allowing users to write programs using common practice for I/O.
 (End of advice to users.)

Like communicators, each file handle has an error handler associated with it. The MPI I/O error handling routines are defined in Section 9.3.

When MPI calls a user-defined error handler resulting from an error on a particular file handle, the first two arguments passed to the file error handler are the file handle and the error code. For I/O errors that are not associated with a valid file handle (e.g., in MPI_FILE_OPEN or MPI_FILE_DELETE), the first argument passed to the error handler is MPI_FILE_NULL.

I/O error handling differs from communication error handling in another important aspect. By default, the predefined error handler for file handles is MPI_ERRORS_RETURN. The **default file error** handler has two purposes: when a new file handle is created (by MPI_FILE_OPEN), the error handler for the new file handle is initially set to the default file error handler, and I/O routines that have no valid file handle on which to raise an error (e.g., MPI_FILE_OPEN or MPI_FILE_DELETE) use the default file error handler. The default file error handler can be changed by specifying MPI_FILE_NULL as the fh argument to MPI_FILE_SET_ERRHANDLER. The current value of the default file error handler can be determined by passing MPI_FILE_NULL as the fh argument to MPI_FILE_GET_ERRHANDLER.

Rationale. For communication, the default error handler is inherited from MPI_COMM_WORLD when using the World Model. In I/O, there is no analogous "root" file handle from which default properties can be inherited. Rather than invent a new global file handle, the default file error handler is manipulated as if it were attached to MPI_FILE_NULL. (*End of rationale.*)

14.8 I/O Error Classes

The implementation dependent error codes returned by the I/O routines can be converted into the error classes defined in Table 14.5.

In addition, calls to routines in this chapter may raise errors in other MPI classes, such as MPI_ERR_TYPE.

14.9 Examples

14.9.1 Double Buffering with Split Collective I/O

This example shows how to overlap computation and output. The computation is performed by the function compute_buffer().

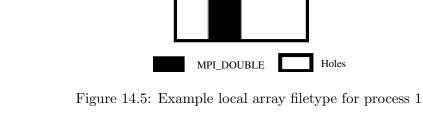
/*=					39
*					40
*	Function:	double_buffer			41
*					42
*	Synopsis:				43
*	void	double_buffer(44
*		MPI_File fh,	**	IN	45
*		MPI_Datatype buftype,	**	IN	46
*		int bufcount	**	IN	47
*)				48

Unofficial Draft for Comment Only

1		Invalid file handle
	MPI_ERR_FILE	
2	MPI_ERR_NOT_SAME	Collective argument not identical on all
3		processes, or collective routines called in
4		a different order by different processes
5	MPI_ERR_AMODE	Error related to the amode passed to
6		MPI_FILE_OPEN
7	MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to
8		MPI_FILE_SET_VIEW
9	MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on
10		a file which supports sequential access only
11	MPI_ERR_NO_SUCH_FILE	File does not exist
12	MPI_ERR_FILE_EXISTS	File exists
13	MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)
14	MPI_ERR_ACCESS	Permission denied
15	MPI_ERR_NO_SPACE	Not enough space
16	MPI_ERR_QUOTA	Quota exceeded
17	MPI_ERR_READ_ONLY	Read-only file or file system
18	MPI_ERR_FILE_IN_USE	File operation could not be completed, as
19		the file is currently open by some process
20	MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-
21		tered because a data representation identi-
22		fier that was already defined was passed to
23		MPI_REGISTER_DATAREP
24	MPI_ERR_CONVERSION	An error occurred in a user supplied data
25		conversion function.
26	MPI_ERR_IO	Other I/O error
27		
28	Table 14.5:	I/O Error Classes
29		
30		
31		
32		
33		
34		
35		
36		
37		
38		
39		
40		
41		
42		
43		
44		
44		
45		
40		
47		
40		

```
1
 *
                                                                              2
 * Description:
                                                                              3
       Performs the steps to overlap computation with a collective write
 *
       by using a double-buffering technique.
 * Parameters:
                          previously opened MPI file handle
 *
       fh
                          MPI datatype for memory layout
       buftype
 *
                                                                              9
                          (Assumes a compatible view has been set on fh)
                                                                              10
 *
       bufcount
                         # buftype elements to transfer
                                                                             11
 *-----
                                                          ----*/
                                                                             12
/* this macro switches which buffer "x" is pointing to */
                                                                             13
                                                                             14
#define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))
                                                                             15
                                                                             16
void double_buffer(MPI_File fh, MPI_Datatype buftype, int bufcount)
                                                                             17
{
                                                                             18
  MPI_Status status; /* status for MPI calls */
                                                                             19
  float *buffer1, *buffer2; /* buffers to hold results */
                                                                             20
  float *compute_buf_ptr; /* destination buffer */
                                                                             21
                             /* for computing */
                                                                             22
                             /* source for writing */
                                                                             23
  float *write_buf_ptr;
                                                                             24
                             /* determines when to quit */
  int done;
                                                                             25
                                                                              26
  /* buffer initialization */
  buffer1 = (float *)
                                                                             27
                     malloc(bufcount*sizeof(float));
                                                                             28
                                                                             29
  buffer2 = (float *)
                    malloc(bufcount*sizeof(float));
                                                                             30
  compute_buf_ptr = buffer1; /* initially point to buffer1 */
                                                                             31
  write_buf_ptr = buffer1;
                              /* initially point to buffer1 */
                                                                             32
                                                                             33
                                                                             34
   /* DOUBLE-BUFFER prolog:
                                                                             35
        compute buffer1; then initiate writing buffer1 to disk
                                                                             36
   *
                                                                             37
   */
   compute_buffer(compute_buf_ptr, bufcount, &done);
                                                                             38
  MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
                                                                             39
                                                                              40
                                                                             41
  /* DOUBLE-BUFFER steady state:
                                                                             42
   * Overlap writing old results from buffer pointed to by write_buf_ptr
   * with computing new results into buffer pointed to by compute_buf_ptr.
                                                                             43
                                                                             44
   * There is always one write-buffer and one compute-buffer in use
                                                                             45
                                                                              46
    * during steady state.
                                                                              47
   */
                                                                              48
  while (!done) {
```

```
1
            TOGGLE_PTR(compute_buf_ptr);
\mathbf{2}
            compute_buffer(compute_buf_ptr, bufcount, &done);
3
            MPI_File_write_all_end(fh, write_buf_ptr, &status);
4
            TOGGLE_PTR(write_buf_ptr);
5
            MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
6
         }
7
8
         /* DOUBLE-BUFFER epilog:
9
               wait for final write to complete.
          *
10
          */
11
         MPI_File_write_all_end(fh, write_buf_ptr, &status);
12
13
14
         /* buffer cleanup */
15
         free(buffer1);
16
         free(buffer2);
17
     }
18
19
     14.9.2 Subarray Filetype Constructor
20
21
22
23
24
25
26
27
28
29
30
^{31}
                                        Process 0
                                                          Process 2
32
                                        Process 1
                                                          Process 3
33
34
                               Figure 14.4: Example array file layout
35
36
37
```



Unofficial Draft for Comment Only

46 47

Assume we are writing out a 100×100 2D array of double precision floating point numbers that is distributed among 4 processes such that each process has a block of 25 columns (e.g., process 0 has columns 0–24, process 1 has columns 25–49, etc.; see Figure 14.4). To create the filetypes for each process one could use the following C program (see Section 5.1.3):

```
double subarray[100][25];
  MPI_Datatype filetype;
  int sizes[2], subsizes[2], starts[2];
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  sizes[0]=100; sizes[1]=100;
   subsizes[0]=100; subsizes[1]=25;
  starts[0]=0; starts[1]=rank*subsizes[1];
  MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C
                            MPI_DOUBLE, &filetype);
   Or, equivalently in Fortran:
double precision subarray(100,25)
integer filetype, rank, ierror
integer sizes(2), subsizes(2), starts(2)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
sizes(1)
            = 100
sizes(2)
            = 100
subsizes(1) = 100
subsizes(2) = 25
starts(1)
            = 0
starts(2) = rank*subsizes(2)
call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
           MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION,
                                                           &
           filetype, ierror)
```

The generated filetype will then describe the portion of the file contained within the process's subarray with holes for the space taken by the other processes. Figure 14.5 shows the filetype created for process 1.

1

 $\mathbf{2}$

3

4

5 6

7

8

9

10 11

12

13

14

15 16

17

18 19

20 21

22

23

24 25

26

27

28

29

30

31

32 33

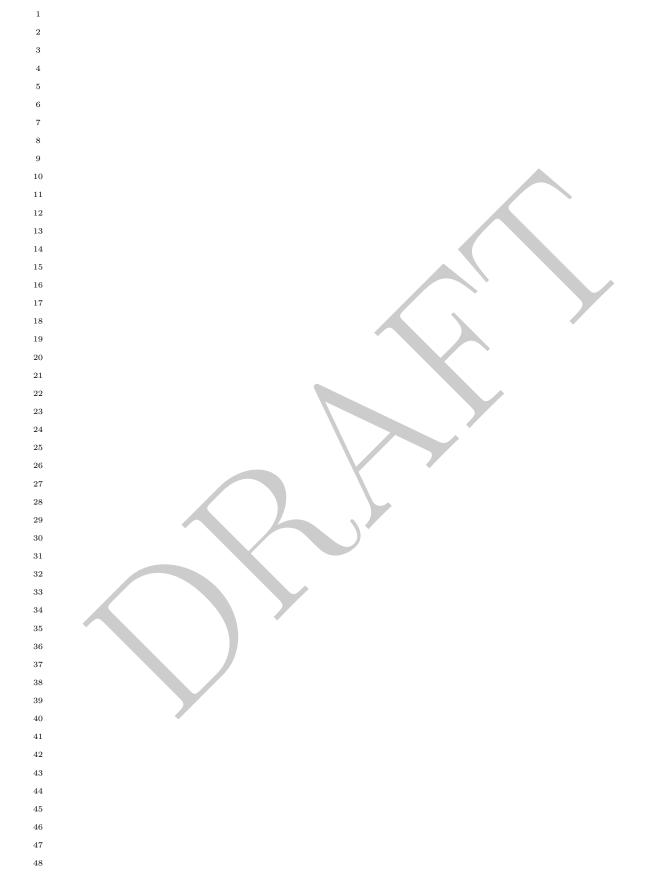
34

35

36 37

38

39



Chapter 15

Tool Support

15.1 Introduction

This chapter discusses interfaces that allow debuggers, performance analyzers, and other tools to extract information about the operation of MPI processes. Specifically, this chapter defines both the MPI profiling interface (Section 15.2), which supports the transparent interception and inspection of MPI calls, and the MPI tool information interface (Section 15.3), which supports the inspection and manipulation of MPI control and performance variables, as well as the registration of callbacks for MPI library events. The interfaces described in this chapter are all defined in the context of an MPI process, i.e., are callable from the same code that invokes other MPI functions.

15.2 Profiling Interface

15.2.1 Requirements

To meet the requirements for the MPI profiling interface, an implementation of the MPI functions must

1. provide a mechanism through which all of the MPI defined functions, except those allowed as macros (See Section 2.6.4), may be accessed with a name shift. This requires, in C and Fortran, an alternate entry point name, with the prefix PMPI_ for each MPI function in each provided language binding and language support method. For routines implemented as macros, it is still required that the PMPI_ version be supplied and work as expected, but it is not possible to replace at link time the MPI_ version with a user-defined version.

For Fortran, the different support methods cause several specific procedure names. Therefore, several profiling routines (with these specific procedure names) are needed for each Fortran MPI routine, as described in Section 19.1.5.

- 2. ensure that those MPI functions that are not replaced may still be linked into an executable image without causing name clashes.
- 3. document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that the profiler developer knows whether she must implement the profile interface for each binding, or can economize by implementing it only for the lowest level routines.

Unofficial Draft for Comment Only

 $45 \\ 46$

4. where the implementation of different language bindings is done through a layered approach (e.g., the Fortran binding is a set of "wrapper" functions that call the C implementation), ensure that these wrapper functions are separable from the rest of the library.

This separability is necessary to allow a separate profiling library to be correctly implemented, since (at least with Unix linker semantics) the profiling library must contain these wrapper functions if it is to perform as expected. This requirement allows the person who builds the profiling library to extract these functions from the original MPI library and add them into the profiling library without bringing along any other unnecessary code.

11 12 13

14

1

2

3

4

5

6

7

8

9

10

5. provide a no-op routine MPI_PCONTROL in the MPI library.

15.2.2 Discussion

The objective of the MPI profiling interface is to ensure that it is relatively easy for authors of profiling (and other similar) tools to interface their codes to MPI implementations on different machines.

Since MPI is a machine independent standard with many different implementations, it is unreasonable to expect that the authors of profiling tools for MPI will have access to the source code that implements MPI on any particular machine. It is therefore necessary to provide a mechanism by which the implementors of such tools can collect whatever performance information they wish *without* access to the underlying implementation.

We believe that having such an interface is important if MPI is to be attractive to end users, since the availability of many different tools will be a significant factor in attracting users to the MPI standard.

The profiling interface is just that, an interface. It says *nothing* about the way in which it is used. There is therefore no attempt to lay down what information is collected through the interface, or how the collected information is saved, filtered, or displayed.

While the initial impetus for the development of this interface arose from the desire to permit the implementation of profiling tools, it is clear that an interface like that specified may also prove useful for other purposes, such as "internetworking" multiple MPI implementations. Since all that is defined is an interface, there is no objection to its being used wherever it is useful.

As the issues being addressed here are intimately tied up with the way in which executable images are built, which may differ greatly on different machines, the examples given below should be treated solely as one way of implementing the objective of the MPI profiling interface. The actual requirements made of an implementation are those detailed in the Requirements section above, the whole of the rest of this section is only present as justification and discussion of the logic for those requirements.

The examples below show one way in which an implementation could be constructed to meet the requirements on a Unix system (there are doubtless others that would be equally valid).

44 45 15.2.3 Logic of the Design

⁴⁶ Provided that an MPI implementation meets the requirements above, it is possible for ⁴⁷ the implementor of the profiling system to intercept the MPI calls that are made by the

user program. She can then collect whatever information she requires before calling the underlying MPI implementation (through its name shifted entry points) to achieve the desired effects.

15.2.4 Miscellaneous Control of Profiling

There is a clear requirement for the user code to be able to control the profiler dynamically at run time. This capability is normally used for (at least) the purposes of

- Enabling and disabling profiling depending on the state of the calculation.
- Flushing trace buffers at non-critical points in the calculation.
- Adding user events to a trace file.

These requirements are met by use of MPI_PCONTROL.

MPI_PCONTROL(level, ...)

IN le

level

Profiling level (integer)

C binding

```
int MPI_Pcontrol(const int level, ...)
```

Fortran 2008 binding

```
MPI_Pcontrol(level)
    INTEGER, INTENT(IN) :: level
```

Fortran binding

MPI_PCONTROL(LEVEL) INTEGER LEVEL

MPI libraries themselves make no use of this routine, and simply return immediately to the user code. However the presence of calls to this routine allows a profiling package to be explicitly called by the user.

Since MPI has no control of the implementation of the profiling code, we are unable to specify precisely the semantics that will be provided by calls to MPI_PCONTROL. This vagueness extends to the number of arguments to the function, and their datatypes.

However to provide some level of portability of user codes to different profiling libraries, we request the following meanings for certain values of level.

- level==0 Profiling is disabled.
- level==1 Profiling is enabled at a normal default level of detail.
- level==2 Profile buffers are flushed, which may be a no-op in some profilers.
- All other values of level have profile library defined effects and additional arguments.

We also request that the default state after MPI has been initialized is for profiling to ⁴⁵ be enabled at the normal default level. (i.e., as if MPI_PCONTROL had just been called ⁴⁶ with the argument 1). This allows users to link with a profiling library and to obtain profile ⁴⁷ output without having to modify their source code at all. ⁴⁸

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

 $\frac{4}{5}$

6

7

8 9

10

11 12

13 14

15 16

17 18

19 20

21

22

23

24

25 26

27

28

29

30

31

32

33

34

35

36

37 38

39 40

41

42 43

The provision of MPI_PCONTROL as a no-op in the standard MPI library supports the
 collection of more detailed profiling information with source code that can still link against
 the standard MPI library.

15.2.5 Profiler Implementation Example

A profiler can accumulate the total amount of data sent by the MPI_SEND function, along with the total elapsed time spent in the function as the following example shows:

```
<sup>10</sup> Example 15.1
```

```
11
     static int totalBytes = 0;
12
     static double totalTime = 0.0;
13
14
     int MPI_Send(const void* buffer, int count, MPI_Datatype datatype,
15
                   int dest, int tag, MPI_Comm comm)
16
     {
17
        double tstart = MPI_Wtime();
                                              /* Pass on all arguments */
18
        int size;
19
                       = PMPI_Send(buffer,count,datatype,dest,tag,comm);
        int result
20
21
        totalTime += MPI Wtime() - tstart;
                                                        /* and time
                                                                               */
22
23
        MPI_Type_size(datatype, &size);
                                            /* Compute size */
24
        totalBytes += count*size;
25
26
        return result;
27
     }
28
29
             MPI Library Implementation Example
     15.2.6
```

If the MPI library is implemented in C on a Unix system, then there are various options,
 including the two presented here, for supporting the name-shift requirement. The choice
 between these two options depends partly on whether the linker and compiler support weak
 symbols.

```
<sup>36</sup> Systems with Weak Symbols
```

³⁷If the compiler and linker support weak external symbols (e.g., Solaris 2.x, other System V.4 machines), then only a single library is required as the following example shows:

4 5

6

7

8 9

30

The effect of this **#pragma** is to define the external symbol MPI_Example as a weak definition. This means that the linker will not complain if there is another definition of the symbol (for instance in the profiling library); however if no other definition exists, then the linker will use the weak definition.

Systems without Weak Symbols

In the absence of weak symbols then one possible solution would be to use the C macro preprocessor as the following example shows:

Example 15.3

```
#ifdef PROFILELIB
# ifdef __STDC__
# define FUNCTION(name) P##name
# else
# define FUNCTION(name) P/**/name
# endif
#else
# define FUNCTION(name) name
#endif
E define function(name) name
#endif
```

Each of the user visible functions in the library would then be declared thus

```
int FUNCTION(MPI_Example)(/* appropriate args */)
{
    /* Useful content */
}
```

The same source file can then be compiled to produce both versions of the library, depending on the state of the PROFILELIB macro symbol.

It is required that the standard MPI library be built in such a way that the inclusion of MPI functions can be achieved one at a time. This is a somewhat unpleasant requirement, since it may mean that each external function has to be compiled from a separate file. However this is necessary so that the author of the profiling library need only define those MPI functions that she wishes to intercept, references to any others being fulfilled by the normal MPI library. Therefore the link step can look something like this

% cc ... -lmyprof -lpmpi -lmpi

Here libmyprof.a contains the profiler functions that intercept some of the MPI functions, libpmpi.a contains the "name shifted" MPI functions, and libmpi.a contains the normal definitions of the MPI functions.

15.2.7 Complications

Multiple Counting

Since parts of the MPI library may themselves be implemented using more basic MPI functions (e.g., a portable implementation of the collective operations implemented using point

Unofficial Draft for Comment Only

 $\mathbf{2}$

 31

 $45 \\ 46$

1 to point communications), there is potential for profiling functions to be called from within $\mathbf{2}$ an MPI function that was called from a profiling function. This could lead to "double 3 counting" of the time spent in the inner routine. Since this effect could actually be useful 4 under some circumstances (e.g., it might allow one to answer the question "How much time $\mathbf{5}$ is spent in the point to point routines when they are called from collective functions?"), we 6 have decided not to enforce any restrictions on the author of the MPI library that would $\overline{7}$ overcome this. Therefore the author of the profiling library should be aware of this problem, 8 and guard against it. In a single-threaded world this is easily achieved through use of a 9 static variable in the profiling code that remembers if you are already inside a profiling 10 routine. It becomes more complex in a multithreaded environment (as does the meaning of 11the times recorded).

12

14

¹³ Linker Oddities

The Unix linker traditionally operates in one pass: the effect of this is that functions from libraries are only included in the image if they are needed at the time the library is scanned. When combined with weak symbols, or multiple definitions of the same function, this can cause odd (and unexpected) effects.

Consider, for instance, an implementation of MPI in which the Fortran binding is 19 achieved by using wrapper functions on top of the C implementation. The author of the 20profile library then assumes that it is reasonable only to provide profile functions for the C 21binding, since Fortran will eventually call these, and the cost of the wrappers is assumed 22 to be small. However, if the wrapper functions are not in the profiling library, then none 23of the profiled entry points will be undefined when the profiling library is called. Therefore 24none of the profiling code will be included in the image. When the standard MPI library 25is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of 26the MPI functions. The overall effect is that the code will link successfully, but will not be 27profiled. 28

To overcome this we must ensure that the Fortran wrapper functions are included in the profiling version of the library. We ensure that this is possible by requiring that these be separable from the rest of the base MPI library. This allows them to be copied out of the base library and into the profiling one using a tool such as **ar**.

33

34 Fortran Support Methods

The different Fortran support methods and possible options for the support of subarrays (depending on whether the compiler can support TYPE(*), DIMENSION(..) choice buffers) imply different specific procedure names for the same Fortran MPI routine. The rules and implications for the profiling interface are described in Section 19.1.5.

 $40 \\ 41$

15.2.8 Multiple Levels of Interception

The scheme given here does not directly support the nesting of profiling functions, since it provides only a single alternative name for each MPI function. Consideration was given to an implementation that would allow multiple levels of call interception, however we were unable to construct an implementation of this that did not have the following disadvantages

 $46 \\ 47$

48

- assuming a particular implementation language,
- imposing a run time cost even when no profiling was taking place.

Since one of the objectives of MPI is to permit efficient, low latency implementations, and it is not the business of a standard to require a particular implementation language, we decided to accept the scheme outlined above.

Note, however, that it is possible to use the scheme above to implement a multi-level system, since the function called by the user may call many different profiling functions before calling the underlying MPI function. This capability has been demonstrated in the P^N MPI tool infrastructure [58].

15.3 The MPI Tool Information Interface

MPI implementations often use internal variables to control their operation and performance and rely on internal events for their implementation. Understanding and manipulating these variables and tracking these events can provide a more efficient execution environment or improve performance for many applications. This section describes the MPI tool information interface, which provides a mechanism for MPI implementors to expose variables, each of which represents a particular property, setting, or performance measurement from within the MPI implementation, as well as expose events that can be tracked by tools. The interface is split into three parts: the first part provides information about, and supports the setting of, control variables through which the MPI implementation tunes its configuration. The second part provides access to performance variables that can provide insight into internal performance information of the MPI implementation. The third part enables tools to query available events within an MPI implementation and register callbacks for them.

To avoid restrictions on the MPI implementation, the MPI tool information interface allows the implementation to specify which control variables, performance variables, and events exist. Additionally, the user of the MPI tool information interface can obtain metadata about each available variable or event, such as its datatype, and a textual description. The MPI tool information interface provides the necessary routines to find all variables and events that exist in a particular MPI implementation; to query their properties; to retrieve descriptions about their meaning; to access and, if appropriate, to alter their values; and (in case of events) set callbacks triggered by them.

Variables, events, and categories across connected MPI processes with equivalent names are required to have the same meaning (see the definition of "equivalent" as related to strings in Section 15.3.3). Furthermore, enumerations with equivalent names across connected MPI processes are required to have the same meaning, but are allowed to comprise different enumeration items. Enumeration items that have equivalent names across connected MPI processes in enumerations with the same meaning must also have the same meaning. In order for variables and categories to have the same meaning, routines in the tools information interface that return details for those variables and categories have requirements on what parameters must be identical. These requirements are specified in their respective sections.

Rationale. The intent of requiring the same meaning for entities with equivalent names is to enforce consistency across connected MPI processes. For example, variables describing the number of packets sent on different types of network devices should have different names to reflect their potentially different meanings. (*End of rationale.*)

The MPI tool information interface can be used independently from the MPI communication functionality. In particular, the routines of this interface can be called before MPI is

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8 9

10 11

12

13

14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40 41

42

43

44

45

 $46 \\ 47$

¹ initialized and after MPI is finalized. In order to support this behavior cleanly, the MPI tool
 ² information interface uses separate initialization and finalization routines. All identifiers
 ³ used in the MPI tool information interface have the prefix MPI_T_.

⁴ On success, all MPI tool information interface routines return MPI_SUCCESS, otherwise ⁵ they return an appropriate and unique return code indicating the reason why the call was not ⁶ successfully completed. Details on return codes can be found in Section 15.3.10. However, ⁷ unsuccessful calls to the MPI tool information interface are not fatal and do not impact the ⁸ execution of subsequent MPI routines.

⁹ Since the MPI tool information interface primarily focuses on tools and support li-¹⁰ braries, MPI implementations are only required to provide C bindings for functions and ¹¹ constants introduced in this section. Except where otherwise noted, all conventions and ¹² principles governing the C bindings of the MPI API also apply to the MPI tool information ¹³ interface, which is available by including the mpi.h header file. All routines in this interface ¹⁴ have local semantics.

Advice to users. The number and type of control variables, performance variables, and events can vary between MPI implementations, platforms and different builds of the same implementation on the same platform as well as between runs. Hence, any application relying on a particular variable will not be portable. Further, there is no guarantee that the number of variables and variable indices are the same across connected MPI processes.

This interface is primarily intended for performance monitoring tools, support tools, and libraries controlling the application's environment. When maximum portability is desired, application programmers should either avoid using the MPI tool information interface or avoid being dependent on the existence of a particular control or performance variable or of a particular event. (*End of advice to users.*)

15.3.1 Verbosity Levels

29The MPI tool information interface provides access to internal configuration and perfor-30 mance information through a set of control and performance variables defined by the MPI 31 implementation. Since some implementations may export a large number of variables, 32 variables are classified by a verbosity level that categorizes both their intended audience 33 (end users, performance tuners or MPI implementors) and a relative measure of level of 34 detail (basic, detailed or all). These verbosity levels are described by a single integer. 35 Table 15.1 lists the constants for all possible verbosity levels. The values of the con-36 stants are monotonic in the order listed in the table; i.e., MPI_T_VERBOSITY_USER_BASIC 37 < MPI_T_VERBOSITY_USER_DETAIL < ... < MPI_T_VERBOSITY_MPIDEV_ALL. 38

39 40

15

16

17

18

19

20

21

22

23

24

25

26 27

28

15.3.2 Binding MPI Tool Information Interface Variables to MPI Objects

 41 Each MPI tool information interface variable provides access to a particular control setting 42or performance property of the MPI implementation. A variable may refer to a specific 43 MPI object such as a communicator, datatype, or one-sided communication window, or the 44variable may refer more generally to the MPI environment of the process. Except for the 45last case, the variable must be bound to exactly one MPI object before it can be used. 46Table 15.2 lists all MPI object types to which an MPI tool information interface variable 47can be bound, together with the matching constant that MPI tool information interface 48routines return to identify the object type.

MPI_T_VERBOSITY_USER_BASIC	Basic information of interest to users	1
MPI_T_VERBOSITY_USER_DETAIL	Detailed information of interest to users	2
MPI_T_VERBOSITY_USER_ALL	All remaining information of interest to users	3
MPI_T_VERBOSITY_TUNER_BASIC	Basic information required for tuning	4
MPI_T_VERBOSITY_TUNER_DETAIL	Detailed information required for tuning	5
MPI_T_VERBOSITY_TUNER_ALL	All remaining information required for tuning	6
MPI_T_VERBOSITY_MPIDEV_BASIC	Basic information for MPI implementors	7
MPI_T_VERBOSITY_MPIDEV_DETAIL	Detailed information for MPI implementors	8
MPI_T_VERBOSITY_MPIDEV_ALL	All remaining information for MPI implementors	9
		10

Constant	MPI object
MPI_T_BIND_NO_OBJECT	N/A; applies globally to entire MPI process
MPI_T_BIND_MPI_COMM	MPI communicators
MPI_T_BIND_MPI_DATATYPE	MPI datatypes
MPI_T_BIND_MPI_ERRHANDLER	MPI error handlers
MPI_T_BIND_MPI_FILE	MPI file handles
MPI_T_BIND_MPI_GROUP	MPI groups
MPI_T_BIND_MPI_OP	MPI reduction operators
MPI_T_BIND_MPI_REQUEST	MPI requests
MPI_T_BIND_MPI_WIN	MPI windows for one-sided communication
MPI_T_BIND_MPI_MESSAGE	MPI message object
MPI_T_BIND_MPI_INFO	MPI info object
MPI_T_BIND_MPI_SESSION	MPI session object

Table 15.1: MPI tool information interface verbosity levels

Table 15.2: Constants to identify associations of variables

Some variables have meanings tied to a specific MPI object. Examples Rationale. include the number of send or receive operations that use a particular datatype, the number of times a particular error handler has been called, or the communication protocol and "eager limit" used for a particular communicator. Creating a new MPI tool information interface variable for each MPI object would cause the number of variables to grow without bound, since they cannot be reused to avoid naming conflicts. By associating MPI tool information interface variables with a specific MPI object, the MPI implementation only must specify and maintain a single variable, which can then be applied to as many MPI objects of the respective type as created during the program's execution. (End of rationale.)

Convention for Returning Strings 15.3.3

Several MPI tool information interface functions return one or more strings. These functions have two arguments for each string to be returned: an OUT parameter that identifies a pointer to the buffer in which the string will be returned, and an INOUT parameter to pass the length of the buffer. The user is responsible for the memory allocation of the buffer and must pass the size of the buffer (n) as the length argument. Let n be the length value specified to the function. On return, the function writes at most n-1 of the string's

Unofficial Draft for Comment Only

 31

¹ characters into the buffer, followed by a null terminator. If the returned string's length is ² greater than or equal to n, the string will be truncated to n-1 characters. In this case, the ³ length of the string plus one (for the terminating null character) is returned in the length ⁴ argument. If the user passes the null pointer as the buffer argument or passes 0 as the ⁵ length argument, the function does not return the string and only returns the length of the ⁶ string plus one in the length argument. If the user passes the null pointer as the length ⁷ argument, the buffer argument is ignored and nothing is returned.

⁸ MPI implementations behave as if they have an internal character array that is copied ⁹ to the output character array supplied by the user. Such output strings are only defined ¹⁰ to be equivalent if their notional source-internal character arrays are identical (up to and ¹¹ including the null terminator), even if the output string is truncated due to a small input ¹² length parameter n.

 $13 \\ 14$

15

15.3.4 Initialization and Finalization

The MPI tool information interface requires a separate set of initialization and finalization routines.

18 19

25

26

MPI_T_INIT_THREAD(required, provided)

1	IN	required	desired level of thread support (integer)
2	OUT	provided	provided level of thread support (integer)
3			

C binding

int MPI_T_init_thread(int required, int *provided)

All programs or tools that use the MPI tool information interface must initialize the 27MPI tool information interface in the processes that will use the interface before calling 28any other of its routines. A user can initialize the MPI tool information interface by calling 29MPI_T_INIT_THREAD, which can be called multiple times. In addition, this routine initial-30 izes the thread environment for all routines in the MPI tool information interface. Calling 31 this routine when the MPI tool information interface is already initialized has no effect 32 beyond increasing the reference count of how often the interface has been initialized. The 33 argument required is used to specify the desired level of thread support. The possible values 34and their semantics are identical to the ones that can be used with MPI_INIT_THREAD 35 listed in Section 11.6. The call returns in provided information about the actual level of 36 thread support that will be provided by the MPI implementation for calls to MPI tool 37 information interface routines. It can be one of the four values listed in Section 11.6. 38

The MPI specification does not require all MPI processes to exist before MPI is initialized. If the MPI tool information interface is used before initialization of MPI, the user is responsible for ensuring that the MPI tool information interface is initialized on all processes it is used in. Processes created by the MPI implementation during initialization inherit the status of the MPI tool information interface (whether it is initialized or not as well as all active sessions and handles) from the process from which they are created.

Processes created at runtime as a result of calls to MPI's dynamic process management
 require their own initialization before they can use the MPI tool information interface.

- 47 48
- Advice to users. If MPI_T_INIT_THREAD is called before MPI_INIT_THREAD, the

requested and provided thread level for MPI_T_INIT_THREAD may influence the behavior and return value of MPI_INIT_THREAD. The same is true for the reverse order. Likewise, when using the Sessions Model (Section 11.3), the requested and provided thread level for MPI_T_INIT_THREAD may influence the behavior and return values of MPI_SESSION_INIT (see Section 11.3), with the same being true for the reverse order. (*End of advice to users.*)

Advice to implementors. MPI implementations should strive to make as many control or performance variables available before MPI initialization (instead of adding them during initialization) to allow tools the most flexibility. In particular, control variables should be available before MPI initialization if their value cannot be changed after MPI initialization. (*End of advice to implementors.*)

MPI_T_FINALIZE()

C binding

int MPI_T_finalize(void)

This routine finalizes the use of the MPI tool information interface and may be called as often as the corresponding MPI_T_INIT_THREAD routine up to the current point of execution. Calling it more times returns a corresponding error code. As long as the number of calls to MPI_T_FINALIZE is smaller than the number of calls to MPI_T_INIT_THREAD up to the current point of execution, the MPI tool information interface remains initialized and calls to its routines are permissible. Further, additional calls to MPI_T_INIT_THREAD after one or more calls to MPI_T_FINALIZE are permissible.

Once MPI_T_FINALIZE is called the same number of times as the routine MPI_T_INIT_THREAD up to the current point of execution, the MPI tool information interface is no longer initialized. The user can reinitialize the interface by a subsequent call to MPI_T_INIT_THREAD.

At the end of the program execution, unless MPI_ABORT is called, an application must have called MPI_T_INIT_THREAD and MPI_T_FINALIZE an equal number of times.

15.3.5 Datatype System

All variables managed through the MPI tool information interface represent their values through typed buffers of a given length and type using an MPI datatype (similar to regular send/receive buffers). Since the initialization of the MPI tool information interface is separate from the initialization of MPI, MPI tool information interface routines can be called before MPI initialization. Consequently, these routines can also use MPI datatypes before MPI initialization. Therefore, within the context of the MPI tool information interface, it is permissible to use a subset of MPI datatypes as specified below before MPI initialization.

Rationale. The MPI tool information interface relies mainly on unsigned datatypes for integer values since most variables are expected to represent counters or resource sizes. MPI_INT is provided for additional flexibility and is expected to be used mainly for control variables and enumeration types (see below).

Providing all basic datatypes, in particular providing all signed and unsigned variants of integer types, would lead to a larger number of types, which tools need to interpret.

Unofficial Draft for Comment Only

1 2

3

4

5 6

7

8

9

10

11

16

17

18 19

20

21

22

23

 24

25

26

27

28

29

30

 31

32 33

34

35 36

37

38

39

40

41

42

43

44

45

1	MPI_INT
2	MPI_INT32_T
3	MPI_INT64_T
4	MPI_UNSIGNED
5	MPI_UNSIGNED_LONG
6	MPI_UNSIGNED_LONG_LONG
7	MPI_UINT32_T
8	MPI_UINT64_T
9	MPI_COUNT
10	MPI_CHAR
11	MPI_DOUBLE
12	
13	
14	Table 15.3: MPI datatypes that can be used by the MPI tool information interface
15	
16	This would cause unnecessary complexity in the implementation of tools based on the
17	MPI tool information interface. (<i>End of rationale.</i>)
18	
19	The MPI tool information interface only relies on a subset of the basic MPI datatypes
20	and does not use any derived MPI datatypes. Table 15.3 lists all MPI datatypes that can
21	be returned by the MPI tool information interface to represent its variables.
22	The use of the datatype MPI_CHAR in the MPI tool information interface implies a null-
23	terminated character array, i.e., a string in the C language. If a variable has type MPI_CHAR,
24	the value of the count parameter returned by MPI_T_CVAR_HANDLE_ALLOC and
25	MPI_T_PVAR_HANDLE_ALLOC must be large enough to include any valid value, including
26	its terminating null character. The contents of returned MPI_CHAR arrays are only defined
27	from index 0 through the location of the first null character.
28	
29	Rationale. The MPI tool information interface requires a significantly simpler type
30	system than MPI itself. Therefore, only its required subset must be present before
31	MPI initialization and MPI implementations do not need to initialize the complete
32	MPI datatype system. (End of rationale.)
33	
34	For variables of type MPI_INT, an MPI implementation can provide additional informa-
35	tion by associating names with a fixed number of values. We refer to this information in
36	the following as an enumeration. In this case, the respective calls that provide additional
37	metadata for each control or performance variable, i.e., MPI_T_CVAR_GET_INFO (Sec-
38	tion 15.3.6), MPI_T_PVAR_GET_INFO (Section 15.3.7), and MPI_T_EVENT_GET_INFO
39	(Section 15.3.8), return a handle of type MPI_T_enum that can be passed to the follow-
40	ing functions to extract additional information. Thus, the MPI implementation can de-
40	scribe variables with a fixed set of values that each represents a particular state. Each
41	enumeration type can have N different values, with a fixed N that can be queried using
42	MPI_T_ENUM_GET_INFO.
43 44	
44 45	
46 47	
47	
40	

CHAPTER 15. TOOL SUPPORT

	OM_GET_INTO(enumype, nu	ini, name, name_ien)	
IN	enumtype	enumeration to be queried (handle)	2
114	enuntype	chumeration to be queried (nandie)	3
OUT	num	number of discrete values represented by this	4
		enumeration (integer)	5
OUT	name	buffer to return the string containing the name of the	6
		enumeration item (string)	7
INOUT	nama lan	length of the string and (on huffor for nome (interes)	8
INCOT	name_len	length of the string and/or buffer for name (integer)	9
			10

MPI_T_ENUM_GET_INFO(enumtype, num, name, name_len)

C binding

If enumtype is a valid enumeration, this routine returns the number of items represented by this enumeration type as well as its name. N must be greater than 0, i.e., the enumeration must represent at least one value.

The arguments name and name_len are used to return the name of the enumeration as described in Section 15.3.3.

The routine is required to return a name of at least length one. This name must be unique with respect to all other names for enumerations that the MPI implementation uses.

Names associated with individual values in each enumeration enumtype can be queried using MPI_T_ENUM_GET_ITEM.

MPI_T_ENUM_	GET_ITEM	(enumtype,	index.	value.	name.	name_len)
	· _ · · _ · · ·	(,			

INI	anumtuna	any manation to be evenied (bandle)	26
IN	enumtype	enumeration to be queried (handle)	27
IN	index	number of the value to be queried in this	28
		enumeration (integer)	29
OUT	value	variable value (integer)	30
OUT	name	buffer to return the string containing the name of the	31
		enumeration item (string)	32
INCLUT			33
INOUT	name_len	length of the string and/or buffer for name (integer)	34
			35

C binding

The arguments name and name_len are used to return the name of the enumeration item as described in Section 15.3.3.

If completed successfully, the routine returns the name/value pair that describes the enumeration at the specified index. The call is further required to return a name of at least length one. This name must be unique with respect to all other names of items for the same enumeration.

15.3.6 **Control Variables**

The routines described in this section of the MPI tool information interface specification focus on the ability to list, query, and possibly set control variables exposed by the MPI implementation. These variables can typically be used by the user to fine tune properties and configuration settings of the MPI implementation. On many systems, such variables can be set using environment variables, although other configuration mechanisms may be available, such as configuration files or central configuration registries. A typical example that is available in several existing MPI implementations is the ability to specify an "eager limit," i.e., an upper bound on the size of messages sent or received using an eager protocol. 10

11 12

1

2

3

4

5

6

7

8

9

Control Variable Query Functions

13 An MPI implementation exports a set of N control variables through the MPI tool infor-14mation interface. If N is zero, then the MPI implementation does not export any control 15variables, otherwise the provided control variables are indexed from 0 to N-1. This index 16number is used in subsequent calls to identify the individual variables.

17An MPI implementation is allowed to increase the number of control variables during 18 the execution of an MPI application when new variables become available through dynamic 19loading. However, MPI implementations are not allowed to change the index of a control 20variable or to delete a variable once it has been added to the set. When a variable becomes 21inactive, e.g., through dynamic unloading, accessing its value should return a corresponding 22error code. 23

Advice to users. While the MPI tool information interface guarantees that indices or variable properties do not change during a particular run of an MPI program, it does not provide a similar guarantee between runs. (End of advice to users.)

The following function can be used to query the number of control variables, num_cvar:

28 29 30

 31 32

33 34

35

36

37

 24

25

26

27

MPI_T_CVAR_GET_NUM(num_cvar)

OUT num_cvar returns number of control variables (integer)

C binding

int MPI_T_cvar_get_num(int *num_cvar)

The function MPI_T_CVAR_GET_INFO provides access to additional information for each variable.

MPI_T_CVAR_GET_INFO(cvar_index, name, name_len, verbosity, datatype, enumtype, desc, desc_len, bind, scope)				
	. ,		3	
IN	cvar_index	index of the control variable to be queried, value between 0 and $num_cvar - 1$ (integer)	4 5	
OUT	name	buffer to return the string containing the name of the control variable (string)	6 7	
INOUT	name_len	length of the string and/or buffer for name (integer)	8	
	_		9	
OUT	verbosity	verbosity level of this variable (integer)	10	
OUT	datatype	MPI data type of the information stored in the	11	
		control variable (handle)	12	
OUT	enumtype	optional descriptor for enumeration information	13	
		(handle)	14	
OUT	desc	buffer to return the string containing a description of	15	
001		the control variable (string)	16	
NIGUT			17	
INOUT	desc_len	length of the string and/or buffer for desc (integer)	18	
OUT	bind	type of MPI object to which this variable must be	19	
		bound (integer)	20	
OUT	scope	scope of when changes to this variable are possible	21	
	·	(integer)	22	
			23 24	
C bindin	C binding			
int MPI T cvar get info(int cvar index char *name int *name len				

After a successful call to MPI_T_CVAR_GET_INFO for a particular variable, subsequent calls to this routine that query information about the same variable must return the same information. An MPI implementation is not allowed to alter any of the returned values.

If any OUT parameter to MPI_T_CVAR_GET_INFO is a NULL pointer, the implementation will ignore the parameter and not return a value for the parameter.

The arguments name and name_len are used to return the name of the control variable as described in Section 15.3.3.

If completed successfully, the routine is required to return a name of at least length one. The name must be unique with respect to all other names for control variables used by the MPI implementation.

The argument verbosity returns the verbosity level of the variable (see Section 15.3.1).

The argument datatype returns the MPI datatype that is used to represent the control variable.

If the variable is of type MPI_INT, MPI can optionally specify an enumeration for the values represented by this variable and return it in enumtype. In this case, MPI returns an enumeration identifier, which can then be used to gather more information as described in Section 15.3.5. Otherwise, enumtype is set to MPI_T_ENUM_NULL. If the datatype is not MPI_INT or the argument enumtype is the null pointer, no enumeration type is returned.

The arguments desc and desc_len are used to return a description of the control variable 47 as described in Section 15.3.3.

Unofficial Draft for Comment Only

26

27

28

29 30

31

32 33

34

35

36 37

38 39

40 41

42

43 44

 $45 \\ 46$

Returning a description is optional. If an MPI implementation does not return a de scription, the first character for desc must be set to the null character and desc_len must
 be set to one at the return of this call.

⁴ The parameter bind returns the type of the MPI object to which the variable must be ⁵ bound or the value MPI_T_BIND_NO_OBJECT (see Section 15.3.2).

6 The scope of a variable determines whether changing a variable's value is either local 7to the MPI process or must be done by the user across multiple connected MPI processes. 8 The latter is further split into variables that require changes in a group of MPI processes 9 and those that require collective changes among all connected MPI processes. Both cases 10can require variables on all participating MPI processes either to be set to consistent (but 11potentially different) values or to equal values. The description provided with the variable 12must contain an explanation about the requirements and/or restrictions for setting the 13particular variable.

On successful return from MPI_T_CVAR_GET_INFO, the argument scope will be set to
 one of the constants listed in Table 15.4.

¹⁶ If the name of a control variable is equivalent across connected MPI processes, the
 ¹⁷ following OUT parameters must be identical: verbosity, datatype, enumtype, bind, and scope.
 ¹⁸ The returned description must be equivalent.

19	F				
20	Scope Constant	Description			
21	MPI_T_SCOPE_CONSTANT	read-only, value is constant			
22	MPI_T_SCOPE_READONLY	read-only, cannot be written, but can change			
23	MPI_T_SCOPE_LOCAL	may be writeable, writing is a local operation			
24	MPI_T_SCOPE_GROUP	may be writeable, must be set to consistent values			
25		across a group of connected MPI processes			
26	MPI_T_SCOPE_GROUP_EQ	may be writeable, must be set to the same value			
27		across a group of connected MPI processes			
28	MPI_T_SCOPE_ALL	may be writeable, must be set to consistent values			
29		across all connected MPI processes			
30	MPI_T_SCOPE_ALL_EQ	may be writeable, must be set to the same value			
31		across all connected MPI processes			
32					
³³ Table 15 4: Scenes for control unrichles					
34	Table 15.4: Scopes for control variables				
35					
36	Advice to users. The scope of a variable only indicates if a variable might be				
37					
38					
39					
40					
41					
42	2 MPI_T_CVAR_GET_INDEX(name, cvar_index)				
43	IN name	name of the control variable (string)			
44	OUT cvar_index	index of the control variable (integer)			
45		mask of the control variable (megor)			
46	C binding				
int MDIT ever get index (const cher trame int tever index)					
48	<pre>int MPI_T_cvar_get_index(const char *name, int *cvar_index)</pre>				

MPI_T_CVAR_GET_INDEX is a function for retrieving the index of a control variable given a known variable name. The name parameter is provided by the caller, and cvar_index is returned by the MPI implementation. The name parameter is a string terminated with a null character.

This routine returns MPI_SUCCESS on success and returns MPI_T_ERR_INVALID_NAME if name does not match the name of any control variable provided by the implementation at the time of the call.

Rationale. This routine is provided to enable fast retrieval of control variables by a tool, assuming it knows the name of the variable for which it is looking. The number of variables exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of variables once at initialization. Although using MPI implementation specific variable names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead at runtime because the tool will not have to iterate over the entire set of variables to find a specific one. (*End of rationale.*)

Example: Printing All Control Variables

Example 15.4 The following example shows how the MPI tool information interface can be used to query and to print the names of all available control variables.

```
#include <stdio.h>
                                                                                       23
                                                                                       ^{24}
#include <stdlib.h>
                                                                                       25
#include <mpi.h>
                                                                                       26
int main(int argc, char *argv[]) {
                                                                                       27
                                                                                       28
  int i, err, num, namelen, bind, verbose, scope;
                                                                                       29
  int threadsupport;
  char name[100];
                                                                                       30
                                                                                       31
  MPI_Datatype datatype;
                                                                                       32
                                                                                       33
  err=MPI_T_init_thread(MPI_THREAD_SINGLE,&threadsupport);
                                                                                       34
  if (err!=MPI_SUCCESS)
                                                                                       35
    return err;
                                                                                       36
                                                                                       37
  err=MPI_T_cvar_get_num(&num);
                                                                                       38
  if (err!=MPI_SUCCESS)
                                                                                       39
    return err;
                                                                                       40
                                                                                       41
  for (i=0; i<num; i++) {</pre>
                                                                                       42
    namelen=100;
    err=MPI_T_cvar_get_info(i, name, &namelen,
                                                                                       43
                                                                                       44
             &verbose, &datatype, NULL,
             NULL, NULL, /*no description */
                                                                                       45
                                                                                       46
             &bind, &scope);
                                                                                       47
    if (err!=MPI_SUCCESS && err!=MPI_T_ERR_INVALID_INDEX) return err;
                                                                                       48
    printf("Var %i: %s\n", i, name);
```

1

 $\mathbf{2}$

3

4

5 6

7

8

9

10

11

12

13

14

15

16 17

18 19 20

21

```
1
        }
2
3
        err=MPI_T_finalize();
4
        if (err!=MPI_SUCCESS)
5
          return 1;
6
        else
7
          return 0;
8
      }
9
10
      Handle Allocation and Deallocation
11
      Before reading or writing the value of a variable, a user must first allocate a handle of type
12
      MPI_T_cvar_handle for the variable by binding it to an MPI object (see also Section 15.3.2).
13
14
            Rationale.
                         Handles used in the MPI tool information interface are distinct from
15
           handles used in the remaining parts of the MPI standard because they must be usable
16
           before MPI is initialized and after MPI is finalized. Further, accessing handles, in
17
           particular for performance variables, can be time critical and having a separate handle
18
           space enables optimizations. (End of rationale.)
19
20
21
      MPI_T_CVAR_HANDLE_ALLOC(cvar_index, obj_handle, handle, count)
22
23
        IN
                  cvar_index
                                                index of control variable for which handle is to be
24
                                                allocated (index)
25
        IN
                  obj_handle
                                                reference to a handle of the MPI object to which this
26
                                                variable is supposed to be bound (pointer)
27
        OUT
                  handle
                                                allocated handle (handle)
28
29
        OUT
                  count
                                                number of elements used to represent this variable
30
                                                (integer)
^{31}
32
      C binding
33
      int MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle,
34
                      MPI_T_cvar_handle *handle, int *count)
35
36
          This routine binds the control variable specified by the argument index to an MPI object.
37
      The object is passed in the argument obj_handle as an address to a local variable that stores
38
      the object's handle. The argument obj_handle is ignored if the MPI_T_CVAR_GET_INFO
      call for this control variable returned MPI_T_BIND_NO_OBJECT in the argument bind. The
39
40
      handle allocated to reference the variable is returned in the argument handle. Upon success-
41
      ful return, count contains the number of elements (of the datatype returned by a previous
42
      MPI_T_CVAR_GET_INFO call) used to represent this variable.
43
           Advice to users. The count can be different based on the MPI object to which the
44
           control variable was bound. For example, variables bound to communicators could
45
           have a count that matches the size of the communicator.
46
47
           It is not portable to pass references to predefined MPI object handles, such as
48
           MPI_COMM_WORLD to this routine, since their implementation depends on the MPI
```

library. Instead, such object handles should be stored in a local variable and the address of this local variable should be passed into MPI_T_CVAR_HANDLE_ALLOC. (*End of advice to users.*)

The value of cvar_index should be in the range 0 to num_cvar -1, where num_cvar is the number of available control variables as determined from a prior call to MPI_T_CVAR_GET_NUM. The type of the MPI object it references must be consistent with the type returned in the bind argument in a prior call to MPI_T_CVAR_GET_INFO.

	51		
			9 10
MPI_T_C	CVAR_HANDLE_FREE(har	ndle)	11
INOUT	handle	handle to be freed (handle)	12
			13
C bindi	ng		14
int MPI	_T_cvar_handle_free(MF	PI_T_cvar_handle *handle)	15
Whe	n a handle is no longer ne	eded, a user of the MPI tool information interface should	16 17
	8	E to free the handle and the associated resources in the	18
MPI imp	lementation. On a succe	essful return, MPI sets the handle to	19
MPI_T_C	VAR_HANDLE_NULL.		20
			21
Control V	ariable Access Functions		22
			23
			24
MPI_I_C	CVAR_READ(handle, buf)		25 26
IN	handle	handle to the control variable to be read (handle)	20
OUT	buf	initial address of storage location for variable value	28
		(choice)	29
			30
C bindi	U U		31
int MPI	_T_cvar_read(MPI_T_cva	ar_handle handle, void *buf)	32
This	routine queries the value of	of a control variable identified by the argument handle and	33 34
stores the	e result in the buffer identi	fied by the parameter buf . The user must ensure that the	34 35
		hold the entire value of the control variable (based on the	36
		prior corresponding calls to MPI_T_CVAR_GET_INFO	37
and MPI.	_T_CVAR_HANDLE_ALLO	JC, respectively).	38
			39
MPI_T_C	CVAR_WRITE(handle, buf)		40
IN	handle	handle to the control variable to be written (handle)	41
IN	buf	initial address of storage location for variable value	42
IIN	bui	(choice)	43 44
			45
C bindi	ng		46
	0	var_handle handle, const void *buf)	47
-			48

Unofficial Draft for Comment Only

1	This routine sets the value of the control variable identified by the argument handle to
2	the data stored in the buffer identified by the parameter buf . The user must ensure that the
3	buffer is of the appropriate size to hold the entire value of the control variable (based on the
4	returned datatype and count from prior corresponding calls to MPI_T_CVAR_GET_INFO
5	and MPI_T_CVAR_HANDLE_ALLOC, respectively).
6	If the variable has a global scope (as returned by a prior corresponding
7	MPI_T_CVAR_GET_INFO call), any write call to this variable must be issued by the user
8	in all connected (as defined in Section 11.10.4) MPI processes. If the variable has group
9	scope, any write call to this variable must be issued by the user in all MPI processes in
10	
11	the group, which must be described by the MPI implementation in the description by the MPI_T_CVAR_GET_INFO.
12	In both cases, the user must ensure that the writes in all participating MPI processes
13	are consistent. If the scope is either MPI_T_SCOPE_ALL_EQ or MPI_T_SCOPE_GROUP_EQ
14	
15	this means that the variable in all connected MPI processes or MPI processes of the group,
16	respectively, must be set to the same value.
17	If it is not possible to change the variable at the time the call is made, the function
18	returns either MPI_T_ERR_CVAR_SET_NOT_NOW, if there may be a later time at which the
19	variable could be set, or MPI_T_ERR_CVAR_SET_NEVER, if the variable cannot be set for the
20	remainder of the application's execution.
20	
21	Example: Reading the Value of a Control Variable
23	
24	Example 15.5 The following example shows a routine that can be used to query the
25	value with a control variable with a given index. The example assumes that the variable is
26	intended to be bound to an MPI communicator.
20	
28	int getValue_int_comm(int index, MPI_Comm comm, int *val) {
29	int err, count;
30	MPI_T_cvar_handle handle;
31	
32	<pre>/* This example assumes that the variable index */</pre>
33	<pre>/* can be bound to a communicator */</pre>
33 34	
	err=MPI_T_cvar_handle_alloc(index, &comm, &handle, &count);
35	if (err!=MPI_SUCCESS) return err;
36 27	
37	/* The following assumes that the variable is $*/$
38	<pre>/* represented by a single integer */</pre>
39	
40	<pre>err=MPI_T_cvar_read(handle,val);</pre>
41	if (err!=MPI_SUCCESS) return err;
42	
43	err=MPI_T_cvar_handle_free(&handle);
44	return err;
45	}
46	
47	
48	

15.3.7 Performance Variables

The following section focuses on the ability to list and to query performance variables provided by the MPI implementation. Performance variables provide insight into MPI implementation specific internals and can represent information such as the state of the MPI implementation (e.g., waiting blocked, receiving, not active), aggregated timing data for submodules, or queue sizes and lengths.

Rationale. The interface for performance variables is separate from the interface for control variables, since performance variables have different requirements and parameters. By keeping them separate, the interface provides cleaner semantics and allows for more performance optimization opportunities. (*End of rationale.*)

Some performance variables and classes refer to *events*. In general, such events describe state transitions within software or hardware related to the performance of an MPI application. The events offered through the callback-driven event-notification interface described in Section 15.3.8 also refer to such state transitions; however, the set of state transitions referred to by performance variables and events as described in Section 15.3.8 may not be identical.

Performance Variable Classes

Each performance variable is associated with a class that describes its basic semantics, possible datatypes, basic behavior, its starting value, whether it can overflow, and when and how an MPI implementation can change the variable's value. The starting value is the value that is assigned to the variable the first time that it is used or whenever it is reset.

Advice to users. If a performance variable belongs to a class that can overflow, it is up to the user to protect against this overflow, e.g., by frequently reading and resetting the variable value. (*End of advice to users.*)

Advice to implementors. MPI implementations should use large enough datatypes for each performance variable to avoid overflows under normal circumstances. (End of advice to implementors.)

The classes are defined by the following constants:

MPI_T_PVAR_CLASS_STATE

A performance variable in this class represents a set of discrete states. Variables of this class are represented by MPI_INT and can be set by the MPI implementation at any time. Variables of this type should be described further using an enumeration, as discussed in Section 15.3.5. The starting value is the current state of the implementation at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

MPI_T_PVAR_CLASS_LEVEL

A performance variable in this class represents a value that describes the utilization ⁴⁵ level of a resource. The value of a variable of this class can change at any time to match ⁴⁶ the current utilization level of the resource. Values returned from variables in this class ⁴⁷ are non-negative and represented by one of the following datatypes: MPI_UNSIGNED, ⁴⁸

Unofficial Draft for Comment Only

 24

MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

• MPI_T_PVAR_CLASS_SIZE

A performance variable in this class represents a value that is the size of a resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG,

MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current size of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

11 12 13

14

15

16

17

18

19

20 21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37 38

39

40

41

42

43

44

45

46 47

48

1

2

3

4

5

6

7

8

9

10

• MPI_T_PVAR_CLASS_PERCENTAGE

The value of a performance variable in this class represents the percentage utilization of a finite resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. It will be returned as an MPI_DOUBLE datatype. The value must always be between 0.0 (resource not used at all) and 1.0 (resource completely used). The starting value is the current percentage utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

• MPI_T_PVAR_CLASS_HIGHWATERMARK

A performance variable in this class represents a value that describes the high watermark utilization of a resource. The value of a variable of this class is non-negative and grows monotonically from the initialization or reset of the variable. It can be represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current utilization level of the resource at the time that the variable is started or reset. MPI implementations must ensure that variables of this class cannot overflow.

MPI_T_PVAR_CLASS_LOWWATERMARK

A performance variable in this class represents a value that describes the low watermark utilization of a resource. The value of a variable of this class is non-negative and decreases monotonically from the initialization or reset of the variable. It can be represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current utilization level of the resource at the time that the variable is started or reset. MPI implementations must ensure that variables of this class cannot overflow.

• MPI_T_PVAR_CLASS_COUNTER

A performance variable in this class counts the number of occurrences of a specific event (e.g., the number of memory allocations within an MPI library). The value of a variable of this class increases monotonically from the initialization or reset of the performance variable by one for each specific event that is observed. Values must be non-negative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG. The starting value for variables of this class is 0. Variables of this class can overflow.

- MPI_T_PVAR_CLASS_AGGREGATE
 - The value of a performance variable in this class is an an aggregated value that

Unofficial Draft for Comment Only

represents a sum of arguments processed during a specific event (e.g., the amount of memory allocated by all memory allocations). This class is similar to the counter class, but instead of counting individual events, the value can be incremented by arbitrary amounts. The value of a variable of this class increases monotonically from the initialization or reset of the performance variable. It must be non-negative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value for variables of this class is 0. Variables of this class can overflow.

• MPI_T_PVAR_CLASS_TIMER

The value of a performance variable in this class represents the aggregated time that the MPI implementation spends executing a particular event, type of event, or section of the MPI library. This class has the same basic semantics as MPI_T_PVAR_CLASS_AGGREGATE, but explicitly records a timing value. The value of a variable of this class increases monotonically from the initialization or reset of the performance variable. It must be non-negative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value for variables of this class is 0. If the type MPI_DOUBLE is used, the units that represent time in this datatype must match the units used by MPI_WTIME. Otherwise, the time units should be documented, e.g., in the description returned by MPI_T_PVAR_GET_INFO. Variables of this class can overflow.

• MPI_T_PVAR_CLASS_GENERIC

This class can be used to describe a variable that does not fit into any of the other classes. For variables in this class, the starting value is variable-specific and implementation-defined.

Performance Variable Query Functions

An MPI implementation exports a set of N performance variables through the MPI tool information interface. If N is zero, then the MPI implementation does not export any performance variables; otherwise the provided performance variables are indexed from 0 to N-1. This index number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of performance variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a performance variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

The following function can be used to query the number of performance variables, num_pvar:

MPI_T_PVAR_GET_NUM(num_pvar)

OUT num_pvar returns number of performance variables (integer)

C binding

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6 7

8 9

10

11

12

13

14

15

16

17

18

19

20

21

22 23

 24

25

26

27 28

29

30 31

32 33

34

35

36

37

38

39

40 41

42 43

44

45

46 47

1	int MPI_7		var)	
2	The function MPI_T_PVAR_GET_INFO provides access to additional information for			
3 4	each varial			
5				
6 7	MPI_T_P\		ame, name_len, verbosity, var_class, datatype, 1, bind, readonly, continuous, atomic)	
8 9 10	IN	pvar_index	index of the performance variable to be queried between 0 and $num_pvar - 1$ (integer)	
11 12	OUT	name	buffer to return the string containing the name of the performance variable (string)	
13	INOUT	name_len	length of the string and/or buffer for name (integer)	
14 15	OUT	verbosity	verbosity level of this variable (integer)	
16	OUT	var_class	class of performance variable (integer)	
17 18	OUT	datatype	MPI datatype of the information stored in the performance variable (handle)	
19 20 21	OUT	enumtype	optional descriptor for enumeration information (handle)	
22 23	OUT	desc	buffer to return the string containing a description of the performance variable (string)	
24	INOUT	desc_len	length of the string and/or buffer for $desc\xspace$ (integer)	
25 26 27	OUT	bind	type of MPI object to which this variable must be bound (integer)	
28 29	OUT	readonly	flag indicating whether the variable can be written/reset (integer)	
30 31	OUT	continuous	flag indicating whether the variable can be started and stopped or is continuously active (integer)	
32 33 34	OUT	atomic	flag indicating whether the variable can be atomically read and reset (integer)	
35	C binding	œ		
36 37			index, char *name, int *name_len,	
38			<pre>*var_class, MPI_Datatype *datatype,</pre>	
39			e, char *desc, int *desc_len, int *bind,	
40		· ·	<pre><continuous, *atomic)<="" int="" pre=""></continuous,></pre>	
41 42			AR_GET_INFO for a particular variable, subsequent	
43		calls to this routine that query information about the same variable must return the same information. An MPI implementation is not allowed to alter any of the returned values.		
44		-	VAR_GET_INFO is a NULL pointer, the implementa-	
45	-		return a value for the parameter.	
46 47 48	The arguments name and name_len are used to return the name of the performance variable as described in Section 15.3.3. If completed successfully, the routine is required to return a name of at least length one.			
	50 100um (a manne of at reast rength one.		

Unofficial Draft for Comment Only

The argument verbosity returns the verbosity level of the variable (see Section 15.3.1). The class of the performance variable is returned in the parameter var_class. The class must be one of the constants defined in Section 15.3.7.

The combination of the name and the class of the performance variable must be unique with respect to all other names for performance variables used by the MPI implementation.

Advice to implementors. Groups of variables that belong closely together, but have different classes, can have the same name. This choice is useful, e.g., to refer to multiple variables that describe a single resource (like the level, the total size, as well as high and low watermarks). (End of advice to implementors.)

The argument datatype returns the MPI datatype that is used to represent the performance variable.

If the variable is of type MPI_INT, MPI can optionally specify an enumeration for the values represented by this variable and return it in enumtype. In this case, MPI returns an enumeration identifier, which can then be used to gather more information as described in Section 15.3.5. Otherwise, enumtype is set to MPI_T_ENUM_NULL. If the datatype is not MPI_INT or the argument enumtype is the null pointer, no enumeration type is returned.

Returning a description is optional. If an MPI implementation does not return a description, the first character for desc must be set to the null character and desc_len must be set to one at the return from this function.

The parameter bind returns the type of the MPI object to which the variable must be bound or the value MPI_T_BIND_NO_OBJECT (see Section 15.3.2).

Upon return, the argument **readonly** is set to zero if the variable can be written or reset by the user. It is set to one if the variable can only be read.

Upon return, the argument **continuous** is set to zero if the variable can be started and stopped by the user, i.e., it is possible for the user to control if and when the value of a variable is updated. It is set to one if the variable is always active and cannot be controlled by the user.

Upon return, the argument **atomic** is set to zero if the variable cannot be read and reset atomically. Only variables for which the call sets **atomic** to one can be used in a call to MPI_T_PVAR_READRESET.

If a performance variable has an equivalent name and has the same class across connected MPI processes, the following OUT parameters must be identical: verbosity, varclass, datatype, enumtype, bind, readonly, continuous, and atomic. The returned description must be equivalent.

MPI_T_PVAR	_GET_INDEX(name,	, var_class,	pvar_index)	
------------	------------------	--------------	-------------	--

IN	name	the name of the performance variable (string)
IN	var_class	the class of the performance variable (integer)
OUT	pvar_index	the index of the performance variable (integer)

C binding

int MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)

MPI_T_PVAR_GET_INDEX is a function for retrieving the index of a performance variable given a known variable name and class. The name and var_class parameters are

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5 6

 $\overline{7}$

8

9

10 11

12

13

14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

 31

32

33

34

35

45

46 47

1 provided by the caller, and pvar_index is returned by the MPI implementation. The name $\mathbf{2}$ parameter is a string terminated with a null character.

3 This routine returns MPI_SUCCESS on success and returns MPI_T_ERR_INVALID_NAME 4 if name does not match the name of any performance variable of the specified var_class provided by the implementation at the time of the call.

Rationale. This routine is provided to enable fast retrieval of performance variables by a tool, assuming it knows the name of the variable for which it is looking. The number of variables exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of variables once at initialization. Although using MPI implementation specific variable names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead at runtime because the tool will not have to iterate over the entire set of variables to find a specific one. (End of rationale.)

16Performance Experiment Sessions

17Within a single program, multiple components can use the MPI tool information interface. 18 To avoid collisions with respect to accesses to performance variables, users of the MPI tool 19 information interface must first create a session. Subsequent calls that access performance 20variables can then be made within the context of this session. Starting, stopping, reading, 21writing, or resetting a variable in one performance experiment session shall not influence 22 whether a variable is started, stopped, read, written, or reset in another performance ex-23periment session. 24

2526

27

33

34 35 36

37

38 39

40

42

43

44

MPI_T_PVAR_SESSION_CREATE(session)

OUT identifier of performance session (handle) session 2829 C binding 30 int MPI_T_pvar_session_create(MPI_T_pvar_session *session) 31 32

This call creates a new session for accessing performance variables and returns a handle for this session in the argument session of type MPI_T_pvar_session.

MPI_T_PVAR_SESSION_FREE(session)

session

identifier of performance experiment session (handle)

C binding

INOUT

int MPI_T_pvar_session_free(MPI_T_pvar_session *session) 41

This call frees an existing session. Calls to the MPI tool information interface can no longer be made within the context of a session after it is freed. On a successful return, MPI sets the session identifier to MPI_T_PVAR_SESSION_NULL.

- 4546
- 47
- 48

5

6 7

8

9

10

11

12

13

Handle Allo	cation and Deallocation		1	
Before using a performance variable, a user must first allocate a handle of type $^{2}_{3}$ MPI_T_pvar_handle for the variable by binding it to an MPI object (see also Section 15.3.2).				
MPLT PV	AR HANDLE ALLOC(session	pvar_index, obj_handle, handle, count)	5 6	
IN IN	session	identifier of performance experiment session (handle)	7	
IN	pvar_index	index of performance experiment session (mandle) be allocated (integer)	8 9 10	
IN	obj_handle	reference to a handle of the MPI object to which this variable is supposed to be bound (pointer)	11 12	
OUT	handle	allocated handle (handle)	13 14	
OUT	count	number of elements used to represent this variable (integer)	15 16 17	
C binding			18	
int MPI_T	_pvar_handle_alloc(MPI_T_ void *obj_handle, MPI	<pre>pvar_session session, int pvar_index, _T_pvar_handle *handle, int *count) e variable specified by the argument index to an</pre>	19 20 21 22	
MPI object in the session identified by the parameter session. The object is passed in the argument obj_handle as an address to a local variable that stores the object's handle. The argument obj_handle is ignored if the MPI_T_PVAR_GET_INFO call for this performance variable returned MPI_T_BIND_NO_OBJECT in the argument bind. The handle allocated to reference the variable is returned in the argument handle. Upon successful return, count contains the number of elements (of the datatype returned by a previous MPI_T_PVAR_GET_INFO call) used to represent this variable. Advice to users. The count can be different based on the MPI object to which the				
	have a count that matches th	For example, variables bound to communicators e size of the communicator.	32 33	
It is not portable to pass references to predefined MPI object handles, such as MPI_COMM_WORLD, to this routine, since their implementation depends on the MPI library. Instead, such an object handle should be stored in a local variable and the address of this local variable should be passed into MPI_T_PVAR_HANDLE_ALLOC. (<i>End of advice to users.</i>)				
The value of index should be in the range 0 to num_pvar - 1, where num_pvar is the number of available performance variables as determined from a prior call to MPI_T_PVAR_GET_NUM. The type of the MPI object it references must be consistent with the type returned in the bind argument in a prior call to MPI_T_PVAR_GET_INFO. For all routines in the rest of this section that take both handle and session as IN or INOUT arguments, if the handle argument passed in is not associated with the session argument, MPI_T_ERR_INVALID_HANDLE is returned.			 39 40 41 42 43 44 45 46 47 48 	

1 MPI_T_PVAR_HANDLE_FREE(session, handle) 2 IN session identifier of performance experiment session (handle) 3 INOUT handle handle to be freed (handle) 4 5C binding 6 int MPI_T_pvar_handle_free(MPI_T_pvar_session session, $\overline{7}$ MPI_T_pvar_handle *handle) 8 9 When a handle is no longer needed, a user of the MPI tool information interface should 10 call MPI_T_PVAR_HANDLE_FREE to free the handle in the session identified by the pa-11 rameter session and the associated resources in the MPI implementation. On a successful 12return, MPI sets the handle to MPI_T_PVAR_HANDLE_NULL. 13 14Starting and Stopping of Performance Variables 1516Performance variables that have the continuous flag set during the query operation are 17continuously operating once a handle has been allocated. Such variables may be queried at 18any time, but they cannot be started or stopped by the user. All other variables are in a 19stopped state after their handle has been allocated; their values are not updated until they have been started by the user. 202122MPI_T_PVAR_START(session, handle) 23 24 IN session identifier of performance experiment session (handle) 25handle of a performance variable (handle) IN handle 2627C binding 28int MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle) 29 30 This functions starts the performance variable with the handle identified by the pa- 31 rameter handle in the session identified by the parameter session. 32 If the constant MPI_T_PVAR_ALL_HANDLES is passed in handle, the MPI implementa-33 tion attempts to start all variables within the session identified by the parameter session for 34which handles have been allocated. In this case, the routine returns MPI_SUCCESS if all vari-35 ables are started successfully (even if there are no non-continuous variables to be started), 36 otherwise MPI_T_ERR_PVAR_NO_STARTSTOP is returned. Continuous variables and vari-37 ables that are already started are ignored when MPI_T_PVAR_ALL_HANDLES is specified. 38 39 MPI_T_PVAR_STOP(session, handle) 4041 IN session identifier of performance experiment session (handle) 42IN handle handle of a performance variable (handle) 43 44C binding 45int MPI_T_pvar_stop(MPI_T_pvar_session session, MPI_T_pvar_handle handle) 4647This functions stops the performance variable with the handle identified by the param-48eter handle in the session identified by the parameter session.

If the constant MPI_T_PVAR_ALL_HANDLES is passed in handle, the MPI implementation attempts to stop all variables within the session identified by the parameter session for which handles have been allocated. In this case, the routine returns MPI_SUCCESS if all variables are stopped successfully (even if there are no non-continuous variables to be stopped), otherwise MPI_T_ERR_PVAR_NO_STARTSTOP is returned. Continuous variables and variables that are already stopped are ignored when MPI_T_PVAR_ALL_HANDLES is specified.

Performance Variable Access Functions

MPI_T_PVAR_READ(session, handle, buf)					
IN	session	identifier of performance experiment session (handle)	13 14		
IN	handle		14 15		
		handle of a performance variable (handle)	16		
OUT	buf	initial address of storage location for variable value	17		
		(choice)	18		
a 1 • 1•			19		
C binding			20		
int MPI_1		ssion session, MPI_T_pvar_handle handle,	21		
	void *buf)		22		
The N	IPI_T_PVAR_READ call quer	ies the value of the performance variable with the	23		
handle han	dle in the session identified by	the parameter session and stores the result in the	24		
	· ·	The user is responsible to ensure that the buffer	25		
		ntire value of the performance variable (based on	26		
		ne corresponding previous calls to	27 28		
		VAR_HANDLE_ALLOC, respectively).	28 29		
	The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the func-				
tion MPI_T_PVAR_READ. 3					
			32		
MPI_T_PV	/AR_WRITE(session, handle, bi	uf)	33		
IN	session	identifier of performance experiment session (handle)	34		
	handle	, , , ,	35		
IN		handle of a performance variable (handle)	36		
IN	buf	initial address of storage location for variable value	37		
		(choice)	38		
			39		
C binding			40		
int MPI_T_pvar_write(MPI_T_pvar_session session, MPI_T_pvar_handle handle, ⁴					
const void *buf) 42					
The MPI_T_PVAR_WRITE call attempts to write the value of the performance variable					
	о ж	ter handle in the session identified by the parameter	45		
session. Th	ne value to be written is passed	l in the buffer identified by the parameter buf . The	46		

Unofficial Draft for Comment Only

user must ensure that the buffer is of the appropriate size to hold the entire value of the per-

 $\mathbf{2}$

 $\overline{7}$

1 2 3 4 5 6 7 8	calls to MF If it is MPI_T_ERF The co	PI_T_PVAR_GET_INFO and N s not possible to change the R_PVAR_NO_WRITE.	e and count returned by the corresponding previous MPI_T_PVAR_HANDLE_ALLOC, respectively). e variable, the function returns NDLES cannot be used as an argument for the func-
9	MPI_T_PV	/AR_RESET(session, handle)	
10	IN	session	identifier of performance experiment session (handle)
11 12	IN	handle	handle of a performance variable (handle)
13 14 15 16			ession session, MPI_T_pvar_handle handle)
16 17 18 19 20 21 22 23 24 25 26	The MPI_T_PVAR_RESET call sets the performance variable with the handle identified by the parameter handle to its starting value specified in Section 15.3.7. If it is not possible to change the variable, the function returns MPI_T_ERR_PVAR_NO_WRITE. If the constant MPI_T_PVAR_ALL_HANDLES is passed in handle, the MPI implementation attempts to reset all variables within the session identified by the parameter session for which handles have been allocated. In this case, the routine returns MPI_SUCCESS if all variables are reset successfully (even if there are no valid handles or all are read-only), otherwise MPI_T_ERR_PVAR_NO_WRITE is returned. Read-only variables are ignored when MPI_T_PVAR_ALL_HANDLES is specified.		
26 27	MPI_T_PV	AR_READRESET(session, har	ndle, buf)
28	IN	session	identifier of performance experiment session (handle)
29 30	IN	handle	handle of a performance variable (handle)
31 32	Ουτ	buf	initial address of storage location for variable value (choice)
33 34 35 36	C binding int MPI_T	g _pvar_readreset(MPI_T_pvar_MPI_T_pvar_handle ha	
 37 38 39 40 41 42 43 44 	This call atomically combines the functionality of MPI_T_PVAR_READ and MPI_T_PVAR_RESET with the same semantics as if these two calls were called separately. If atomic operations on this variable are not supported, this routine returns MPI_T_ERR_PVAR_NO_ATOMIC. The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the function MPI_T_PVAR_READRESET.		
44 45 46 47 48	inform form	mation interface, in particular ance variables, from any prog	g-based tools rely on the ability to call the MPI tool r routines to start, stop, read, write, and reset per- ram context, including asynchronous contexts such tations should strive, if possible in their particular

15.3. THE MPI TOOL INFORMATION INTERFACE

environment, to enable these usage scenarios for all or a subset of the routines mentioned above. If implementing only a subset, the read, write, and reset routines are typically the most critical for sampling based tools. An MPI implementation should clearly document any restrictions on the program contexts in which the MPI tool information interface can be used. Restrictions might include guaranteeing usage outside of all signals or outside a specific set of signals. Any restrictions could be documented, for example, through the description returned by MPI_T_PVAR_GET_INFO. (End of advice to implementors.)

Rationale. All routines to read, to write or to reset performance variables require the session argument. This requirement keeps the interface consistent and allows the use of MPI_T_PVAR_ALL_HANDLES where appropriate. Further, this opens up additional performance optimizations for the implementation of handles. (End of rationale.)

Example: Tool to Detect Receives with Long Unexpected Message Queues

Example 15.6 The following example shows a sample tool to identify receive operations that occur during times with long message queues. This examples assumes that the MPI implementation exports a variable with the name "MPI_T_UMQ_LENGTH" to represent the current length of the unexpected message queue. The tool is implemented as a PMPI tool using the MPI profiling interface.

The tool consists of three parts: (1) the initialization (by intercepting the call to MPI_INIT), (2) the test for long unexpected message queues (by intercepting calls to MPI_RECV), and (3) the clean-up phase (by intercepting the call to MPI_FINALIZE). To capture all receives, the example would have to be extended to have similar wrappers for all receive operations.

Part 1 — Initialization: During initialization, the tool searches for the variable and, once the right index is found, allocates a session and a handle for the variable with the found index, and starts the performance variable.

<pre>#include <stdio.h></stdio.h></pre>	32
<pre>#include <stdlib.h></stdlib.h></pre>	33
<pre>#include <string.h></string.h></pre>	34
<pre>#include <assert.h></assert.h></pre>	35
<pre>#include <mpi.h></mpi.h></pre>	36
	37
/* Global variables for the tool */	38
<pre>static MPI_T_pvar_session session;</pre>	39
<pre>static MPI_T_pvar_handle handle;</pre>	40
	41
<pre>int MPI_Init(int *argc, char ***argv) {</pre>	42
<pre>int err, num, i, index, namelen, verbosity;</pre>	43
<pre>int var_class, bind, threadsup;</pre>	44
int readonly, continuous, atomic, count;	45
char name[18];	46
MPI_Comm comm;	47
MPI_Datatype datatype;	48

 $\overline{7}$

 24

```
1
           MPI_T_enum enumtype;
2
3
           err=PMPI_Init(argc, argv);
4
           if (err!=MPI_SUCCESS) return err;
5
6
           err=PMPI_T_init_thread(MPI_THREAD_SINGLE, &threadsup);
7
           if (err!=MPI_SUCCESS) return err;
8
9
           err=PMPI_T_pvar_get_num(&num);
10
           if (err!=MPI_SUCCESS) return err;
11
           index=-1;
12
           i=0;
13
           while ((i<num) && (index<0) && (err==MPI_SUCCESS)) {</pre>
14
                 /* Pass a buffer that is at least one character longer than */
15
                 /* the name of the variable being searched for to avoid */
16
                 /* finding variables that have a name that has a prefix */
17
                 /* equal to the name of the variable being searched. */
18
                 namelen=18;
19
                  err=PMPI_T_pvar_get_info(i, name, &namelen, &verbosity,
20
                          &var_class, &datatype, &enumtype, NULL, NULL, &bind,
21
                          &readonly, &continuous, &atomic);
22
                  if (strcmp(name,"MPI_T_UMQ_LENGTH")==0) index=i;
23
                  i++; }
24
           if (err!=MPI_SUCCESS) return err;
25
26
           /* this could be handled in a more flexible way for a generic tool */
27
           assert(index>=0);
           assert(var_class==MPI_T_PVAR_CLASS_LEVEL);
28
29
           assert(datatype==MPI_INT);
30
           assert(bind==MPI_T_BIND_MPI_COMM);
31
32
           /* Create a session */
33
           err=PMPI_T_pvar_session_create(&session);
34
           if (err!=MPI_SUCCESS) return err;
35
36
           /* Get a handle and bind to MPI_COMM_WORLD */
37
           comm=MPI_COMM_WORLD;
38
           err=PMPI_T_pvar_handle_alloc(session, index, &comm, &handle, &count);
39
           if (err!=MPI_SUCCESS) return err;
40
41
           /* this could be handled in a more flexible way for a generic tool */
42
           assert(count==1);
43
44
           /* Start variable */
45
           err=PMPI_T_pvar_start(session, handle);
46
           if (err!=MPI_SUCCESS) return err;
47
48
           return MPI_SUCCESS;
```

}

Part 2 — Testing the Queue Lengths During Receives: During every receive operation, the tool reads the unexpected queue length through the matching performance variable and compares it against a predefined threshold.

```
#define THRESHOLD 5
```

int value, err;

```
{
```

}

```
if (comm==MPI_COMM_WORLD) {
    err=PMPI_T_pvar_read(session, handle, &value);
    if ((err==MPI_SUCCESS) && (value>THRESHOLD))
    {
        /* tool identified receive called with long UMQ */
        /* execute tool functionality, */
        /* e.g., gather and print call stack */
    }
}
```

return PMPI_Recv(buf, count, datatype, source, tag, comm, status);

Part 3 — Termination: In the wrapper for MPI_FINALIZE, the MPI tool information interface is finalized.

```
int MPI_Finalize(void)
{
    int err;
    err=PMPI_T_pvar_handle_free(session, &handle);
    err=PMPI_T_pvar_session_free(&session);
    err=PMPI_T_finalize();
    return PMPI_Finalize();
}
```

15.3.8 Events

During the execution of an MPI application, the MPI implementation can raise events of a specific type to inform the user of a state change in the implementation. Event types describe specific state changes within the MPI implementation. In comparison to aggregate performance variables, events provide per-instance information on such state changes. The MPI implementation is said to *raise an event* when it invokes a callback function previously registered for the corresponding event type by the user. Each callback invocation for a specific event instance has a timestamp associated with it, which can be queried by the user, describing the time when the event was observed by the implementation. This decouples

Unofficial Draft for Comment Only

the observation of the state change from the communication of this information to the user.
 A timestamp in this context is a count of clock ticks elapsed since some time in the past
 and represented as a variable of type MPI_Count.

Event Sources

As a means to manage multiple state changes to be observed concurrently by different parts of the software and hardware system, the event interface of the MPI Tool Information Interface uses the concept of *sources*. A source in this context is a concept describing the logical entity raising the event. A source may or may not directly represent a concrete part of the software or hardware system. This concept is used primarily to describe partial ordering of events across different components where total ordering cannot necessarily be determined or is too costly to enforce.

14

4 5

6

- The following function can be used to query the number of event sources, *num_sources*:
- $15 \\ 16$

17

19

21

MPI_T_SOURCE_GET_NUM(num_sources)

num_sources

18 OUT

returns number of event sources (integer)

²⁰ C binding

int MPI_T_source_get_num(int *num_sources)

The number of available event sources can be queried with a call to MPI_T_SOURCE_GET_NUM. An MPI implementation is allowed to increase the number of sources during the execution of an MPI process. However, MPI implementations are not allowed to change the index of an event source or to delete an event source once it has been made visible to the user (e.g., if new event sources become available via dynamic loading of additional components in the MPI implementation).

- 29 30
- 30 31
- 32
- 33
- 34

48

INsource_indexindex of the source to be queried between 0 and num_sources - 1 (integer)3OUTnamebuffer to return the string containing the name of the source (string)6OUTname_lenlength of the string and/or buffer for name (integer)8OUTdescbuffer to return the string containing the description of the source (string)9INOUTdesc_lenlength of the string and/or buffer for desc (integer)12OUTorderingflag indicating chronological ordering guarantees given by the source (integer)13OUTticks_per_secondthe number of ticks per second for the timer of this source (integer)16OUTmax_ticksthe maximum count of ticks reported by this source before overflow occurs (integer)18OUTinfooptional info object (handle)20	MPI_T_SOURCE_GET_INFO(source_index, name, name_len, desc, desc_len, ordering, ticks_per_second, max_ticks, info) ²				
OUTnamebuffer to return the string containing the name of the source (string)5INOUTname_lenlength of the string and/or buffer for name (integer)8OUTdescbuffer to return the string containing the description of the source (string)9INOUTdesc_lenlength of the string and/or buffer for desc (integer)11OUTorderingflag indicating chronological ordering guarantees given by the source (integer)13OUTticks_per_secondthe number of ticks per second for the timer of this 	IN	source_index	index of the source to be queried between 0 and		
OUTnamebuffer to return the string containing the name of the source (string)6 7INOUTname_lenlength of the string and/or buffer for name (integer)8OUTdescbuffer to return the string containing the description of the source (string)9INOUTdesc_lenlength of the string and/or buffer for desc (integer)12OUTordering given by the source (integer)13OUTticks_per_secondthe number of ticks per second for the timer of this source (integer)15OUTmax_ticksthe maximum count of ticks reported by this source before overflow occurs (integer)18OUTinfooptional info object (handle)20			$num_sources - 1 (integer)$		
OUTdescbuffer to return the string containing the description of the source (string)9INOUTdesc_lenlength of the string and/or buffer for desc (integer)11OUTorderingflag indicating chronological ordering guarantees given by the source (integer)13OUTticks_per_secondthe number of ticks per second for the timer of this source (integer)15OUTmax_ticksthe maximum count of ticks reported by this source before overflow occurs (integer)17OUTinfooptional info object (handle)20	OUT	name		6	
OUTdescbuffer to return the string containing the description of the source (string)10INOUTdesc_lenlength of the string and/or buffer for desc (integer)12OUTorderingflag indicating chronological ordering guarantees given by the source (integer)13OUTticks_per_secondthe number of ticks per second for the timer of this source (integer)15OUTmax_ticksthe maximum count of ticks reported by this source before overflow occurs (integer)17OUTinfooptional info object (handle)20	INOUT	name_len	length of the string and/or buffer for name (integer)	8	
INOUT desc_len length of the string and/or buffer for desc (integer) 11 OUT ordering flag indicating chronological ordering guarantees given by the source (integer) 13 OUT ticks_per_second the number of ticks per second for the timer of this source (integer) 16 OUT max_ticks the maximum count of ticks reported by this source before overflow occurs (integer) 17 OUT info optional info object (handle) 20	OUT	desc	buffer to return the string containing the description	9	
INOUT desc_len length of the string and/or buffer for desc (integer) 12 OUT ordering flag indicating chronological ordering guarantees 13 OUT ordering flag indicating chronological ordering guarantees 13 OUT ticks_per_second the number of ticks per second for the timer of this source (integer) 15 OUT max_ticks the maximum count of ticks reported by this source before overflow occurs (integer) 18 OUT info optional info object (handle) 20	001			10	
OUT ticks_per_second 14 OUT ticks_per_second the number of ticks per second for the timer of this source (integer) 15 OUT max_ticks the maximum count of ticks reported by this source before overflow occurs (integer) 17 OUT info optional info object (handle) 20	INOUT	desc_len			
OUT ticks_per_second 14 OUT ticks_per_second the number of ticks per second for the timer of this source (integer) 15 OUT max_ticks the maximum count of ticks reported by this source before overflow occurs (integer) 17 OUT info optional info object (handle) 20	OUT	ordering	flag indicating chronological ordering guarantees	13	
OU1 ticks_per_second the number of ticks per second for the timer of this source (integer) 16 OUT max_ticks the maximum count of ticks reported by this source is before overflow occurs (integer) 18 OUT info optional info object (handle) 20		<u> </u>		14	
OUT max_ticks source (integer) 16 OUT max_ticks the maximum count of ticks reported by this source 18 before overflow occurs (integer) 19 OUT info optional info object (handle) 20	OUT	ticks per second	the number of ticks per second for the timer of this	15	
OUTmax_ticksthe maximum count of ticks reported by this source before overflow occurs (integer)18OUTinfooptional info object (handle)20				16	
OUT info before overflow occurs (integer) 19 20	OUT	max ticks	the maximum count of ticks reported by this source		
OUTinfooptional info object (handle)20	001				
-		info			
	001	inio	optional into object (nandie)	20 21	

C binding

<pre>int MPI_T_source_get_info(int source_index, char *name, int *name_len,</pre>
<pre>char *desc, int *desc_len, MPI_T_source_order *ordering,</pre>
<pre>MPI_Count *ticks_per_second, MPI_Count *max_ticks,</pre>
MPI_Info *info)

A call to MPI_T_SOURCE_GET_INFO returns additional information on the source identified by the source_index argument.

The arguments name and name_len are used to return the name of the source as described in Section 15.3.3.

The arguments desc and desc_len are used to return the description of the source as described in Section 15.3.3.

The ordering argument returns whether event callbacks of this source will be invoked in chronological order, i.e., the timestamps reported by MPI_T_EVENT_GET_TIMESTAMP of subsequent events of the same source are monotonically increasing. The value of ordering can be MPI_T_SOURCE_ORDERED or MPI_T_SOURCE_UNORDERED.

The ticks_per_seconds argument returns the number of ticks elapsed in one second for the timer used for the specific source.

The max_ticks argument returns the largest number of ticks reported by this source as a timestamp before the value overflows.

Advice to users. As the size of MPI_Count is defined in relation to the types MPI_Aint and MPI_Offset, the effective size of MPI_Count may lead to overflows of the timestamp values reported. Users can use the argument max_ticks to mitigate resulting problems. (End of advice to users.)

MPI can optionally return an info object containing the default hints set for this source. If the argument to info provided by the user is the NULL pointer, this argument is ignored,

Unofficial Draft for Comment Only

28

29

30

31

32

33

34

35

36

37

38

39

40

41 42

43

44

45

4647

otherwise an MPI implementation is required to return all hints that are supported by
 the implementation for this source and have default values specified; any user-supplied
 hints that were not ignored by the implementation; and any additional hints that were
 set by the implementation. If no such hints exist, a handle to a newly created info object
 is returned that contains no key/value pair. The user is responsible for freeing info via
 MPI_INFO_FREE.

MPI_T_SOURCE_GET_TIMESTAMP(source_index, timestamp)
 IN source_index index of the source (integer)
 OUT timestamp current timestamp from specified source (integer)

C binding

14 15 16

17

23

30

31

32

33

34

35 36 37

38 39

40

 41

13

8

int MPI_T_source_get_timestamp(int source_index, MPI_Count *timestamp)

To enable proper query of a reference timestamp for a specific source, a user can obtain a current timestamp using MPI_T_SOURCE_GET_TIMESTAMP. The argument

source_index identifies the index of the source to query. The call returns MPI_SUCCESS and
 a current timestamp in the argument timestamp if the source supports ad-hoc generation of
 timestamps. The call returns MPI_T_ERR_INVALID_INDEX if the index does not identify a
 valid source. The call returns MPI_T_ERR_NOT_SUPPORTED if the source does not support
 the ad-hoc generation of timestamps.

²⁴ ₂₅ Callback Safety Requirements

The actions a user is allowed to perform inside a callback function may vary with its execution context. As the user has no control over the execution context of specific callback function invocations, MPI provides a way to communicate this information using callback safety levels.

> Safety Requirement MPI_T_CB_REQUIRE_NONE MPI_T_CB_REQUIRE_MPI_RESTRICTED MPI_T_CB_REQUIRE_THREAD_SAFE MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE

Table 15.5: Hierarchy of safety requirement levels for event callback routines.

Table 15.5 provides the hierarchy of callback safety requirements levels within userdefined callback functions. The MPI implementation provides the safety requirement as an argument to the callback when it is invoked.

⁴² The level of MPI_T_CB_REQUIRE_NONE is the lowest level and does not impose any ⁴³ restrictions on the callback function.

The level of MPI_T_CB_REQUIRE_MPI_RESTRICTED restricts the set of MPI functions
 that can be called from inside the callback to all functions with the prefix MPI_T as well as
 MPI_WTICK and MPI_WTIME.

- 47
- 48

Advice to users. While some MPI functions are safe to be called inside a callback

function used in the MPI tool information interface—which may in some implementations be issued from asynchronous contexts such as signal handlers—this does not imply that those MPI functions are generally safe to be called in asynchronous contexts such as signal handlers. (*End of advice to users.*)

The level of MPI_T_CB_REQUIRE_THREAD_SAFE includes all the limitations of MPI_T_CB_REQUIRE_MPI_RESTRICTED and additionally requires the callback to be reentrant and thread-safe. This means the callback must allow its execution to be interrupted by or happen concurrently with any other callback including itself.

The level of MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE includes all the limitations of MPI_T_CB_REQUIRE_THREAD_SAFE and additionally requires the callback to meet the safety requirements needed to support invocations from asynchronous contexts, such as signal handlers.

Advice to users. It is always safe to assume the highest restrictions for a callback invocation (i.e., MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE). By evaluating the specific requirements at runtime, a tool may obtain more freedom of action within the callback. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will strive to set callback safety requirements to the most permissive level for a given callback invocation. (*End of advice to implementors.*)

All functions with the prefix MPI_T, except those listed in Table 15.6, may return the error code MPI_T_ERR_NOT_ACCESSIBLE to indicate that the user may not access this function at this time.

MPI_T_EVENT_COPY	PMPI_T_EVENT_COPY
MPI_T_EVENT_GET_SOURCE	PMPI_T_EVENT_GET_SOURCE
MPI_T_EVENT_GET_TIMESTAMP	PMPI_T_EVENT_GET_TIMESTAMP
MPI_T_EVENT_READ	PMPI_T_EVENT_READ
MPI_T_PVAR_READ	PMPI_T_PVAR_READ
MPI_T_PVAR_READRESET	PMPI_T_PVAR_READRESET
MPI_T_PVAR_RESET	PMPI_T_PVAR_RESET
MPI_T_PVAR_START	PMPI_T_PVAR_START
MPI_T_PVAR_STOP	PMPI_T_PVAR_STOP
MPI_T_PVAR_WRITE	PMPI_T_PVAR_WRITE
MPI_T_SOURCE_GET_TIMESTAMP	PMPI_T_SOURCE_GET_TIMESTAMP

Table 15.6: List of MPI functions that when called from within a callback function may not return MPI_T_ERR_NOT_ACCESSIBLE.

Rationale. A call may be implemented in a way that is not safe for all execution contexts of a callback function, e.g., inside a signal handler. An MPI implementation therefore needs a way to communicate its inability to perform a certain action due to the execution context of a callback invocation. (*End of rationale.*)

 $\mathbf{2}$

 $\overline{7}$

 24

A high-quality implementation shall not return Advice to implementors. MPI_T_ERR_NOT_ACCESSIBLE except where absolutely necessary. (End of advice to *implementors.*) Advice to users. Users intercepting calls into the MPI tool information interface using the PMPI interface must ensure that the safety requirements for the calling context are met. This means that users may have to implement the wrapper with the highest safety level used by the MPI implementation. (End of advice to users.) **Event Type Query Functions** An MPI implementation exports a set of N event types through the MPI tool information interface. If N is zero, then the MPI implementation does not export any event types; otherwise, the provided event types are indexed from 0 to N-1. This index number is used in subsequent calls to identify a specific event type. An MPI implementation is allowed to increase the number of event types during the execution of an MPI process. However, MPI implementations are not allowed to change the index of an event type or to delete an event type once it has been made visible to the user (e.g., if new event types become available via dynamic loading of additional components in the MPI implementation). The following function can be used to query the number of event types, *num_events*: MPI_T_EVENT_GET_NUM(num_events) 24 OUT returns number of event types (integer) num_events C binding int MPI_T_event_get_num(int *num_events) The function MPI_T_EVENT_GET_INFO provides access to additional information about a specific event type. 31

MPI_T_EVENT_GET_INFO(event_index, name, name_len, verbosity, array_of_datatypes, array_of_displacements, num_elements, enumtype, info, desc, desc_len, bind)

	IN	event_index	index of the event type to be queried between 0 and $num_events - 1$ (integer)	4 5 6
	OUT	name	buffer to return the string containing the name of the event type (string)	7 8
	INOUT	name_len	length of the string and/or buffer for name (integer)	9
	OUT	verbosity	verbosity level of this event type (integer)	10
	OUT	array_of_datatypes	array of MPI basic datatypes used to encode the event data (array of handles)	11 12 13
	OUT	array_of_displacements	array of byte displacements of the elements in the event buffer (array of non-negative integers)	14 15
	INOUT	num_elements	length of array_of_datatypes and array_of_displacements arrays (non-negative integer)	16 17 18
	OUT	enumtype	optional descriptor for enumeration information (handle)	19 20
	OUT	info	optional info object (handle)	21
	OUT	desc	buffer to return the string containing a description of	22 23
	INOUT	desc_len	the event type (string) length of the string and/or buffer for desc (integer)	$24 \\ 25$
	OUT	bind	type of MPI object to which an event of this type	26
			must be bound (integer)	27
				28
r	! hinding			29

C binding

int MPI_T_eve	<pre>nt_get_info(int event_index, char *name, int *name_len,</pre>
	<pre>int *verbosity, MPI_Datatype array_of_datatypes[],</pre>
	<pre>MPI_Aint array_of_displacements[], int *num_elements,</pre>
	<pre>MPI_T_enum *enumtype, MPI_Info *info, char *desc,</pre>
	<pre>int *desc_len, int *bind)</pre>

After a successful call to MPI_T_EVENT_GET_INFO for a particular event type, subsequent calls to this routine that query information about the same event type must return the same information. If any INOUT or OUT argument to MPI_T_EVENT_GET_INFO is a NULL pointer, the implementation will ignore the argument and not return a value for the specific argument.

The arguments name and name_len are used to return the name of the event type as described in Section 15.3.3. If completed successfully, the routine is required to return a name of at least length one. The name of the event type must be unique with respect to all other names for event types used by the MPI implementation.

The argument verbosity returns the verbosity level of the event type (see Section 15.3.1).

The argument array_of_datatypes returns an array of MPI datatype handles that de-46scribe the elements returned for an instance of the event type with index event_index. The 47event data can either be queried element by element with MPI_T_EVENT_READ or copied 48

Unofficial Draft for Comment Only

1 $\mathbf{2}$

3

36

37

38

39

40

41

42

43

44

1 into a contiguous event buffer with MPI_T_EVENT_COPY. For the latter case, the argu- $\mathbf{2}$ ment array_of_displacements returns an array of byte displacements in the event buffer in 3 ascending order starting with zero.

The user is responsible for the memory allocation for the array_of_datatypes and

 $\mathbf{5}$ array_of_displacements arrays. The number of elements in each array is supplied by the user 6 in num_elements. If the number of elements used by the event type is larger than the value 7of num_elements provided by the user, the number of datatype handles and displacements 8 returned in the corresponding arrays is truncated to the value of num_elements passed in 9 by the user. If the user passes the NULL pointer for array_of_datatypes or

10 array_of_displacements, the respective arguments are ignored. Unless the user passes the 11NULL pointer for num_elements, the function returns the number of elements required for 12this event type. If the user passes the NULL pointer for num_elements, the arguments 13num_elements, array_of_datatypes, and array_of_displacements are ignored.

14MPI can optionally return an enumeration identifier in the enumtype argument, de-15scribing the individual elements in the array_of_datatypes argument. Otherwise, enumtype 16is set to MPI_T_ENUM_NULL. If the argument to enumtype provided by the user is the 17MPI_T_ENUM_NULL pointer, no enumeration type is returned.

18 MPI can optionally return an info object containing the default hints set for a regis-19tration handle for this event type. If the argument to info provided by the user is the NULL 20pointer, this argument is ignored, otherwise an MPI implementation is required to return 21all hints that are supported by the implementation for a registration handle for this event 22type and have default values specified; any user-supplied hints that were not ignored by the 23implementation; and any additional hints that were set by the implementation. If no such 24 hints exist, a handle to a newly created info object is returned that contains no key/value 25pair. The user is responsible for freeing info via MPI_INFO_FREE.

26The arguments desc and desc_len are used to return the description of the event type as described in Section 15.3.3. Returning a description is optional. If an MPI implementation 2728 does not return a description, the first character for desc must be set to the null character 29and desc_len must be set to one at the return from this function.

30 The parameter bind returns the type of the MPI object to which the event type must 31 be bound or the value MPI_T_BIND_NO_OBJECT (see Section 15.3.2).

32 If an event type has an equivalent name across connected MPI processes, the following 33 OUT parameters must be identical: verbosity, array_of_datatypes, num_elements, enumtype, 34and bind. The returned description must be equivalent. As the argument

35 array_of_displacements is process dependent, it may differ across connected MPI processes. 36 This routine returns MPI_SUCCESS on success and returns MPI_T_ERR_INVALID_INDEX 37 if event_index does not match a valid event type index provided by the implementation at 38the time of the call.

39 40

41

MPI_T_EVENT_GET_INDEX(name, event_index)

42	IN	name	name of the event type (string)
43 44	OUT	event_index	index of the event type (integer)
45			

C binding 46

```
int MPI_T_event_get_index(const char *name, int *event_index)
47
48
```

MPI_T_EVENT_GET_INDEX returns the index of an event type identified by a known event type name. The name parameter is provided by the caller, and event_index is returned by the MPI implementation. The name parameter is a string terminated with a null character.

This routine returns MPI_SUCCESS on success and returns MPI_T_ERR_INVALID_NAME if **name** does not match the name of any event type provided by the implementation at the time of the call.

Rationale. This routine is provided to enable fast retrieval of an event index by a tool, assuming it knows the name of the event type for which it is looking. The number of event types exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of event types once at initialization. Although using MPI implementation specific event type names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead at runtime because the tool will not have to iterate over the entire set of event types to find a specific one. (*End of rationale.*)

Handle Allocation and Deallocation

Before the MPI implementation calls a callback function on the occurrence of a specific event, the user needs to register a callback function to be called for that event type and obtain a handle of type MPI_T_event_registration.

MPI_T_I	MPI_T_EVENT_HANDLE_ALLOC(event_index, obj_handle, info, event_registration)			
IN	event_index	index of event type for which the registration handle	25 26	
		is to be allocated (integer)	27	
IN	obj_handle	reference to a handle of the MPI object to which this	28	
		event is supposed to be bound (pointer)	29	
IN	info	info object (handle)	30	
OUT	event_registration	event registration (handle)	31	
001	event_registration	event registration (nandie)	32	
			33	

C binding

MPI_T_EVENT_HANDLE_ALLOC creates a *registration handle* for the event type identified by event_index. Furthermore, if required by the event type, the registration handle is bound to the object referred to by the argument obj_handle. The argument obj_handle is ignored if the MPI_T_EVENT_GET_INFO call for this event type returned MPI_T_BIND_NO_OBJECT in the argument bind. The user can pass hints for the handle allocation to the MPI implementation via the info argument. The allocated event-registration handle is returned in the argument event_registration.

1

 $\mathbf{2}$

3

4

5 6

7

8

9

10

11

12

13

14

15

16 17

18 19

20

21

22 23

34

35

36 37

38

39

40

41

42

43

1	MPI_T_EV	ENT_HANDLE_SET_INFO	event_registration, info)
2 3	IN	event_registration	event registration (handle)
4	IN	info	info object (handle)
5 6 7 8	C binding int MPI_T		
9 10 11 12 13 14 15 16	dle associa no effect o effect on p MPI imple: Advi	T_EVENT_HANDLE_SET_II ted with event_registration n previously set or defaulted hi reviously set or defaulted hi mentation in this call to MF ce to users. Some info ite	The second secon
17 18 19 20 21 22	is cre woul unab MPI_	eated. Thus, an implemend have accepted in an hand ble to update certain info hing T_EVENT_HANDLE_GET_	tation may ignore hints issued in this call that it lle allocation call. An implementation may also be its in a call to MPI_T_EVENT_HANDLE_SET_INFO. INFO can be used to determine whether info changes tion. (<i>End of advice to users.</i>)
23 24			
25	MPI_T_EV	/ENT_HANDLE_GET_INFO	event_registration, info_used)
26 27	IN	event_registration	event registration (handle)
28	OUT	info_used	info object (handle)
29 30 31 32 33	C binding int MPI_T	_event_handle_get_info	ration event_registration,
34 35 36 37 38 39 40 41 42 43 44	the event-relation hints related is required values spectary addition to a newly	registration handle associate ed to this registration hand to return all hints that are cified; any user-supplied hin onal hints that were set by	O returns a new info object containing the hints of ed with event_registration. The current setting of all le is returned in info_used. An MPI implementation is supported by the implementation and have default its that were not ignored by the implementation; and the implementation. If no such hints exist, a handle rned that contains no key/value pairs. The user is PI_INFO_FREE.
45 46 47 48			

MPI_T_E	MPI_T_EVENT_REGISTER_CALLBACK(event_registration, cb_safety, info, user_data,			
	event_cb_function) · · · ·	2	
IN	event_registration	event registration (handle)	3	
IIN	event_registration	event registration (nandre)	4	
IN	cb_safety	maximum callback safety level (integer)	5	
IN	info	info object (handle)	6	
	user data	n sinten ta a norm aantusllad haffan	7	
IN	user_data	pointer to a user-controlled buffer	8	
IN	event_cb_function	pointer to user-defined callback function (function)	9	
			10	

C binding

int MPI_T_event_register_callback(

MPI_T_event_registration event_registration, MPI_T_cb_safety cb_safety, MPI_Info info, void *user_data, MPI_T_event_cb_function event_cb_function)

MPI_T_EVENT_REGISTER_CALLBACK associates a user-defined function pointed to by event_cb_function with an allocated event-registration handle. The maximum callback safety level supported by the callback function is passed in the argument cb_safety. The safety levels are defined in Table 15.5. A user can register multiple callback functions for a given event-registration handle, potentially specifying one for each callback safety level. Registering a callback function for a specific callback safety level overwrites any previously registered callback function pointer and info object associated with the event registration for the specific callback safety level. If event_cb_function is the NULL pointer, the association of a callback function for that callback safety level is removed.

When an event is triggered, the implementation will select from all registered callbacks the callback with the lowest safety level valid in the context in which the callback is invoked. In situations where the required callback safety level exceeds the highest level for which a callback function is registered for a given registration handle, the event instance is dropped.

At invocation time, the implementation passes the pointer to a user-defined memory region specified during callback registration with the argument user_data.

The user can pass hints for the registration of the specified callback function to the MPI implementation via the info argument.

Advice to users. As event instances can be raised as soon as the registration handle is associated with the first callback function, the callback function with the highest callback safety guarantees should be registered before any further registrations for lower callback safety guarantees, to avoid dropped events due to insufficient callback safety guarantees. (*End of advice to users.*)

The callback function passed to MPI_T_EVENT_REGISTER_CALLBACK in the argument event_cb_function needs to have the following type:

```
typedef void (*MPI_T_event_cb_function)(
```

MPI_T_event_instance event_instance, MPI_T_event_registration event_registration, MPI_T_cb_safety cb_safety, void *user_data);

Unofficial Draft for Comment Only

11

12

13

14

15 16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

31

32

33 34

35

36

37

38

39 40

41

42 43

44

45

46

47

1 2 3 4 5 6 7 8 9 10 11 12	object of invocation to the even function to identify th even to de describes to The argun	type MPI_T_event_instance. T of the function to which it is p at-registration handle returned to the same event type and bou e specific event registration in allocate the handle from within the safety requirements the call	sponds to a handle for the opaque event-instance this handle is only valid inside the corresponding assed. The argument event_registration corresponds d by MPI_T_EVENT_HANDLE_ALLOC for the user and object combination. The handle can be used to formation, such as event type and bound object, or in the callback invocation. The argument cb_safety lback function must fulfill in the current invocation. o user-allocated memory that was passed to the MPI tion.		
13	MPI_T_E\	/ENT_CALLBACK_SET_INFO	(event_registration, cb_safety, info)		
14	IN	event_registration	event registration (handle)		
15	IN	cb_safety	callback safety level (integer)		
16 17	IN	info	info object (handle)		
18					
20 21 22 23 24 25 26 27 28 29	C binding int MPI_T_event_callback_set_info(
30 21					
31 32			(event_registration, cb_safety, info_used)		
33	IN	event_registration	event registration (handle)		
34	IN	cb_safety	callback safety level (integer)		
35 36	OUT	info_used	info object (handle)		
37 38 39 40	C binding int MPI_T_event_callback_get_info(
41 42 43 44 45 46 47 48	of the call cb_safety of setting of returned i supported	EVENT_CALLBACK_GET_INF black function registered for to of the event-registration hand all hints related to this callban n info_used. An MPI implem by the implementation and ha	••••••••••••••••••••••••••••••••••••••		

1 the implementation. If no such hints exist, a handle to a newly created info object is $\mathbf{2}$ returned that contains no key/value pairs. The user is responsible for freeing info_used via 3 MPI_INFO_FREE. 4 To stop the MPI implementation from raising events for a specific registration, a user needs to free the corresponding event-registration handle. 56 7 MPI_T_EVENT_HANDLE_FREE(event_registration, user_data, free_cb_function) 8 9 event registration (handle) IN event_registration 10 IN user_data pointer to a user-controlled buffer 11 IN free_cb_function pointer to user-defined callback function (function) 1213 C binding 14int MPI_T_event_handle_free(MPI_T_event_registration event_registration, 15void *user_data, 16MPI_T_event_free_cb_function free_cb_function) 1718 MPI_T_EVENT_HANDLE_FREE returns MPI_SUCCESS when deallocation of the handle 19 was initiated successfully and returns MPI_T_ERR_INVALID_HANDLE if 20event_registration does not match a valid allocated event-registration handle at the time 21of the call. The callback function free_cb_function is called by the MPI implementation, 22when it is able to guarantee that no further event instances for the corresponding event-23registration handle will be raised. If the pointer to free_cb_function is the NULL pointer, no 24 user function is invoked after successful deallocation of the event registration handle. The 25pointer to user-controlled memory provided in the user_data argument will be passed to the 26function provided in the free cb function on invocation. 27Advice to users. A free-callback function associated with a registration handle should 2829always be prepared to postpone any pending actions, should the provided callback safety requirements exceed those required by the pending actions. (End of advice to 30 31users.) 32 The callback function passed to MPI_T_EVENT_HANDLE_FREE in the argument 33 free_cb_function needs to have the following type: 34 35typedef void (*MPI_T_event_free_cb_function)(36 MPI_T_event_registration event_registration, 37 MPI_T_cb_safety cb_safety, 38 void *user_data); 39

Handling Dropped Events

42Events may occur at times when the MPI implementation cannot invoke the user function corresponding to a matching event handle. An implementation is allowed to buffer such 4344events and delay the callback invocation. If an event occurs at times when the corresponding callback function cannot be called and the corresponding data cannot be buffered, or no callback function meeting the required callback safety level is registered, the event data may be dropped. To discover such data loss, the user can set a handler function for a specific event-registration handle.

Unofficial Draft for Comment Only

40

41

45

46

47

	120		
1	MPI_T_E	VENT_SET_DROPPED_HA	NDLER(event_registration, dropped_cb_function)
3	IN	event_registration	valid event registration (handle)
4	IN	dropped_cb_function	pointer to user-defined callback function (function)
5			
6	C bindi	ng	
7	int MPI_	T_event_set_dropped_han	dler(
8		MPI_T_event_regist	ration event_registration,
9		MPI_T_event_droppe	ed_cb_function dropped_cb_function)
10	MPL	T_EVENT_SET_DROPPED	_HANDLER registers the function
11			the MPI implementation when event information is
12 13	dropped	for the registration handle s	pecified in event_registration. Subsequent calls to
14			NDLER with the same registration handle will replace
15	*		ns for that registration handle. If the pointer to
16		_	tter, no data loss is recorded or reported until a new
17	valid call	back function is registered.	
18	Add	vice to users. The invocation	on of the dropped handler callback function may not
19			at was actually lost. (End of advice to users.)
20			
21		-	MPI_T_EVENT_SET_DROPPED_HANDLER in the
22 23	argument	t dropped_cb_function needs	to have the following type:
24	typedef	<pre>void (*MPI_T_event_drop</pre>	<pre>ped_cb_function)(int count,</pre>
25	01	-	vent_registration event_registration,
26		int sou	rce_index,
27			b_safety cb_safety,
28		void *u	ser_data);
29	The	argument event registration	corresponds to the event registration handle to which
30			argument count provides a best effort estimation of
31 32			ered event callback corresponding to event_registration
33	that were	e not executed since the reg	istration of the dropped-callback handler or the last
34		9	allback handler. The source_index provides the index
35			sponding event information. The argument cb_safety
36			callback function must fulfill in the current invocation.
37	-		described in Table 15.5. The argument user_data is
38	-	registration.	v that was passed to the MPI implementation during
39	Camback .		
40	Add	vice to users. A callback fu	nction for dropped events associated with a registra-
41 42			repared to postpone any pending actions, should the
42	-		ements exceed those required by the pending actions.
44	(Er	nd of advice to users.)	
45	Ads	vice to implementors. A	high-quality implementation will strive to invoke a
46		-	events associated with a registration handle at times
47			f action to the function as possible. (End of advice to
48	imp	olementors.)	

If events are dropped for a specific source, the corresponding handler callback function must be called before other events are raised for this source. This means in a sequence of five events E1 to E5 from the same source, where E3 and E4 were dropped, any handler function set through MPI_T_EVENT_SET_DROPPED_HANDLER for event-registration handles associated with E3 or E4 must be called before E5 is raised.

Reading Event Data

In event callbacks, the parameter event_instance provides access to the per-instance event data, i.e., the data encoded by the specific event type for this instance. The user can obtain event data as well as event meta data, such as a time stamp and the source, by providing this handle to the respective query functions. The event-instance handle is invalid beyond the scope of the current invocation of the callback function to which it is provided.

The callback function argument event_registration identifies the registration handle that was used to register the callback function.

The callback function argument cb_safety indicates the requirements for the specific callback invocation. The value is one of the safety requirements levels described in Table 15.5. The argument user_data passes the pointer provided by the user during callback registration back to the function call.

Advice to users. Depending on the registered event and usage of MPI by the application, a callback function may be invoked with high frequency. Users should therefore strive to minimize the amount of work done inside callback functions. Furthermore, the time spent in a callback function may influence the capability of an implementation to buffer events and long execution times may lead to an increased number of dropped events. (*End of advice to users.*)

MPI provides the following function calls to access data of a specific event instance and its corresponding meta data (such as its time and source).

MPI_T_EVENT	_READ(eve	nt_instance,	element_	_index,	buffer)
-------------	-----------	--------------	----------	---------	---------

	·	,	32
IN	event_instance	event-instance handle provided to the callback function (handle)	33
		function (nancie)	34
1N	element_index	index into the array of datatypes of the item to be	35
		queried (integer)	36
OUT	buffer	pointer to a memory location to store the item data	37
		(choice)	38
			39
~			40

C binding

MPI_T_EVENT_READ allows users to copy one element of the event data to a userspecified buffer at a time.

The event_instance argument identifies the event instance to query. It is erroneous 46 to provide any other event-instance handle to the call than the one passed by the MPI 47 implementation to the callback function in which the data is read. The buffer argument 48

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

1 must point to a memory location the MPI implementation can copy the element of the event $\mathbf{2}$ data to identified by element_index. 3 4 MPI_T_EVENT_COPY(event_instance, buffer) 56 IN event_instance event instance provided to the callback function 7 (handle) 8 OUT buffer user-allocated buffer for event data (choice) 9 10 C binding 11 int MPI_T_event_copy(MPI_T_event_instance event_instance, void *buffer) 1213 MPI_T_EVENT_COPY copies the event data as a whole into the user-provided buffer. 14The user must assure that the buffer is of at least the size of the extent of the event 15type, which can be computed from the type and displacement information returned by 16the corresponding call to MPI_T_EVENT_GET_INFO. The data may include padding bytes 17between individual elements of the event data in the buffer. A user can reconstruct the 18location and size of the data contained in the buffer through the information returned by 19MPI_T_EVENT_GET_INFO. 20Advice to implementation. An implementation should strive to use an appropriately 2122 compact representation when copying event instance data to a user buffer via 23MPI_T_EVENT_COPY to reduce the amount of memory required for the user buffer. 24 (End of advice to implementors.) 2526Reading Event Meta Data 27Additional to the specific event data encoded by each event type, supplemental information 28available across all event types can be queried. 29 30 31MPI_T_EVENT_GET_TIMESTAMP(event_instance, event_timestamp) 32 IN event_instance event instance provided to the callback function 33 (handle) 3435OUT event_timestamp timestamp the event was observed (integer) 36 37 C binding 38 int MPI_T_event_get_timestamp(MPI_T_event_instance event_instance, 39 MPI_Count *event_timestamp) 40 MPI_T_EVENT_GET_TIMESTAMP returns the timestamp of when the event was ini-41 tially observed by the implementation. The event_instance argument identifies the event 42instance to query. It is erroneous to provide any other handle to the call than the one 43 passed by the MPI implementation to the callback function in which the timestamp is read. 4445Advice to users. An MPI implementation may postpone the call to the user's callback 46function. In this case, the call to MPI_T_EVENT_GET_TIMESTAMP may yield a 47 timestamp in the past that is closer to the time the event was initially observed, as 48

opposed to a timestamp captured during callback function invocation. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will return a timestamp as close as possible to the earliest time the event was observed by the MPI implementation. (*End of advice to implementors.*)

An event may be raised from different components acting as event sources in the MPI implementation. A source in this context is an abstract concept that helps to define partial ordering of raised events, as each source provides its own ordering guarantees. A source describes the entity that raises the event, rather than the origin of the data.

To identify the source of an event instance, the user can query the index of the source within the corresponding event callback function invocation.

Advice to implementors. An excessive number of event sources may negatively impact performance of a tool due to per-source overhead in event handling. (End of advice to implementors.)

MPI_T_EVENT_GET_SOURCE(event_instance, source_index)					
IN	event_instance	event instance provided to the callback function			
		(handle)			
OUT	source_index	index identifying the source (integer)			

C binding

The event_instance argument identifies the event instance to query. It is erroneous to provide any other event-instance handle to the call than the one passed by the MPI implementation to the callback function in which the source is queried.

The source_index argument returns the index of the source of the event instance. It can be used to query more information on the source using MPI_T_SOURCE_GET_INFO.

Rationale. Event callback function invocations are associated with a source to enable chronological processing of events on the tool side, when required, while retaining low overhead on the side of the MPI implementation. (*End of rationale.*)

15.3.9 Variable Categorization

MPI implementations can optionally group performance and control variables into categories to express logical relationships between various variables. For example, an MPI implementation could group all control and performance variables that refer to message transfers in the MPI implementation and thereby distinguish them from variables that refer to local resources such as memory allocations or other interactions with the operating system.

Categories can also contain other categories to form a hierarchical grouping. Categories ⁴⁶ can never include themselves, either directly or transitively within other included categories. ⁴⁷ Expanding on the example above, this allows MPI to refine the grouping of variables referring ⁴⁸

Unofficial Draft for Comment Only

 31

to message transfers into variables to control and to monitor message queues, message
 matching activities and communication protocols. Each of these groups of variables would
 be represented by a separate category and these categories would then be listed in a single
 category representing variables for message transfers.

⁵ The category information may be queried in a fashion similar to the mechanism for ⁶ querying variable information. The MPI implementation exports a set of N categories via ⁷ the MPI tool information interface. If N = 0, then the MPI implementation does not export ⁸ any categories, otherwise the provided categories are indexed from 0 to N - 1. This index ⁹ number is used in subsequent calls to functions of the MPI tool information interface to ¹⁰ identify the individual categories.

¹¹ An MPI implementation is permitted to increase the number of categories during the ¹² execution of an MPI program when new categories become available through dynamic load-¹³ ing. However, MPI implementations are not allowed to change the index of a category or ¹⁴ delete it once it has been added to the set.

¹⁵ Similarly, MPI implementations are allowed to add variables to categories, but they
 ¹⁶ are not allowed to remove variables from categories or change the order in which they are
 ¹⁷ returned.

¹⁹ Category Query Functions

The following function can be used to query the number of categories, num_cat.

MPI_T_CATEGORY_GET_NUM(num_cat)

OUT num_cat current number of categories (integer)

C binding

- int MPI_T_category_get_num(int *num_cat)
- 29 30 31

18

20

21 22 23

 24

25 26

27

Individual category information can then be queried by calling the following function:

MPI_T_CA	TEGORY_GET_INFO(cat_ind num_pvars, num_categor	ex, name, name_len, desc, desc_len, num_cvars, ies)	1 2		
IN	cat_index	index of the category to be queried (integer)	3 4		
OUT	name	buffer to return the string containing the name of the category (string)	4 5 6		
INOUT	name_len	length of the string and/or buffer for name (integer)	7		
OUT	desc	buffer to return the string containing the description of the category (string)	8 9 10		
INOUT	desc_len	length of the string and/or buffer for $desc$ (integer)	11		
OUT	num_cvars	number of control variables in the category (integer)	12		
OUT	num_pvars	number of performance variables in the category (integer)	13 14 15		
Ουτ	num_categories	number of categories contained in the category (integer)	16 17		
			18		
•	C binding int MPI_T_category_get_info(int cat_index, char *name, int *name_len, char *desc, int *desc_len, int *num_cvars, int *num_pvars, int *num_categories) 19 20 21 22 22				
described i The re- unique wit If any plementati The a described i Return description be set to o The fu- categories num_categ If the turned des	The arguments name and name_len are used to return the name of the category as described in Section 15.3.3. The routine is required to return a name of at least length one. This name must be unique with respect to all other names for categories used by the MPI implementation. If any OUT parameter to MPI_T_CATEGORY_GET_INFO is the NULL pointer, the im- plementation will ignore the parameter and not return a value for the parameter. The arguments desc and desc_len are used to return the description of the category as described in Section 15.3.3. Returning a description is optional. If an MPI implementation decides not to return a description, the first character for desc must be set to the null character and desc_len must be set to one at the return of this call. The function returns the number of control variables, performance variables and other categories, respectively. If the name of a category is equivalent across connected MPI processes, then the re- turned description must be equivalent.				
MPI_T_CA	TEGORY_GET_NUM_EVENT	S(cat_index, num_events)	41		
IN	cat_index	index of the category to be queried (integer)	42 43		
OUT	num_events	number of event types in the category (integer)	44		
C binding int MPI_T	45 C binding nt MPI_T_category_get_num_events(int cat_index, int *num_events) 47				
	48				

Unofficial Draft for Comment Only

m the qu	eried category.	Γ_NUM_EVENTS returns the number of event types contained.
MPI_T_C	ATEGORY_GET_INI	DEX(name, cat_index)
IN	name	the name of the category (string)
OUT	cat_index	the index of the category (integer)
C bindini int MPI_	-	ndex(const char *name, int *cat_index)
given a k is returne null chara This	nown category name, ed by the MPI impler acter. routine returns MPI, loes not match the n	[_INDEX is a function for retrieving the index of a cate . The name parameter is provided by the caller, and cat_ nentation. The name parameter is a string terminated w _SUCCESS on success and returns MPI_T_ERR_INVALID_N name of any category provided by the implementation a
Dy	a toor, assuming it k	
num pos Alt MP at n to f	nber of categories exp sible for the tool to s hough using MPI im I implementations, to	posed by the implementation can change over time, so it is simply iterate over the list of categories once at initializate plementation specific category names is not portable at bol developers may choose to take this route for lower over tool will not have to iterate over the entire set of categories End of rationale.)
num pos Alt MP at n to f	nber of categories exp sible for the tool to s hough using MPI im l implementations, to runtime because the ind a specific one. (A Member Query Funct	ions
num pos Alt MP at n to f Category MPI_T_C	nber of categories exp sible for the tool to s hough using MPI im I implementations, to runtime because the ind a specific one. (<i>I</i> Member Query Funct CATEGORY_GET_CV	posed by the implementation can change over time, so it is simply iterate over the list of categories once at initializate plementation specific category names is not portable a pol developers may choose to take this route for lower over tool will not have to iterate over the entire set of categories <i>End of rationale.</i>) cions
num pos Alt MP at n to f	nber of categories exp sible for the tool to s hough using MPI im l implementations, to runtime because the ind a specific one. (A Member Query Funct	posed by the implementation can change over time, so it is simply iterate over the list of categories once at initializate plementation specific category names is not portable a pol developers may choose to take this route for lower over tool will not have to iterate over the entire set of categories End of rationale.)
num pos Alt MP at n to f Category MPI_T_C	nber of categories exp sible for the tool to s hough using MPI im I implementations, to runtime because the ind a specific one. (<i>I</i> Member Query Funct CATEGORY_GET_CV	posed by the implementation can change over time, so it is simply iterate over the list of categories once at initializate plementation specific category names is not portable a pol developers may choose to take this route for lower over tool will not have to iterate over the entire set of categor <i>End of rationale.</i>) cions
num pos Alt MP at n to f Category MPI_T_C IN	nber of categories exp sible for the tool to s hough using MPI im l implementations, to cuntime because the ind a specific one. (<i>I</i> Member Query Funct CATEGORY_GET_CV cat_index	<pre>bosed by the implementation can change over time, so it is simply iterate over the list of categories once at initialization applementation specific category names is not portable a bol developers may choose to take this route for lower over tool will not have to iterate over the entire set of catego End of rationale.)</pre>
num pos Alt MP at n to f Category MPI_T_C IN IN OUT	nber of categories exp sible for the tool to s hough using MPI im l implementations, to cuntime because the ind a specific one. (<i>I</i> Member Query Funct CATEGORY_GET_CV cat_index len indices	<pre>bosed by the implementation can change over time, so it is simply iterate over the list of categories once at initialization plementation specific category names is not portable a bol developers may choose to take this route for lower over tool will not have to iterate over the entire set of catego End of rationale.)</pre>

MPI_T_CA	ATEGORY_GET_PVARS(cat_ir	ndex, len, indices)	1		
IN	cat_index	index of the category to be queried, in the range 0 and $num_cat - 1$ (integer)	2 3		
IN	len	the length of the indices array (integer)	4 5		
OUT	indices	an integer array of size len, indicating performance	6		
		variable indices (array of integers)	7		
			8		
C binding	g		9		
int MPI_7	[_category_get_pvars(int of the second sec	<pre>cat_index, int len, int indices[])</pre>	10 11		
MPI_	T_CATEGORY_GET_PVARS	an be used to query which performance variables	12		
		A category contains zero or more performance	13		
variables.			14		
			15		
MPI_T_CA	ATEGORY_GET_EVENTS(cat_	index, len, indices)	16		
IN	cat_index	index of the category to be queried, in the range 0	17 18		
		and $num_cat - 1$ (integer)	19		
IN	len	the length of the indices array (integer)	20		
OUT	indices	an integer array of size len, indicating event type	21 22		
		indices (array of integers)	22		
			24		
C binding					
<pre>int MPI_T_category_get_events(int cat_index, int len, int indices[]) 20</pre>					
MPI_	T_CATEGORY_GET_EVENTS	can be used to query which event types are con-	27		
tained in a	a particular category. A catego	bry contains zero or more event types.	28 29		
			30		
MPI_T_CA	ATEGORY_GET_CATEGORIES	;(cat_index, len, indices)	31		
IN	cat_index	index of the category to be queried, in the range 0	32		
IN		and num_cat -1 (integer)	33		
IN	len	the length of the indices array (integer)	34		
			35		
OUT	indices	an integer array of size len, indicating category indices (array of integers)	36 37		
		indices (array of integers)	38		
C binding	σ		39		
	0	(int cat_index, int len, int indices[])	40		
			41		
		RIES can be used to query which other categories	42		
are contained in a particular category. A category contains zero or more other categories. As mentioned above, MPI implementations can grow the number of categories as well 44					
as the number of variables or other categories within a category. In order to allow users 45					
		check quickly whether new categories have been	46		
added or	new variables or categories h	ave been added to a category, MPI maintains a	47		

Unofficial Draft for Comment Only

1 virtual timestamp. This timestamp is monotonically increasing during the execution and is $\mathbf{2}$ returned by the following function: 3 4 MPI_T_CATEGORY_CHANGED(stamp) 56 OUT a virtual time stamp to indicate the last change to stamp 7 the categories (integer) 8 9 C binding 10 int MPI_T_category_changed(int *stamp) 11 If two subsequent calls to this routine return the same timestamp, it is guaranteed that 12the category information has not changed between the two calls. If the timestamp retrieved 13 from the second call is higher, then some categories have been added or expanded. 1415Advice to users. The timestamp value is purely virtual and only intended to check 16for changes in the category information. It should not be used for any other purpose. 17 (End of advice to users.) 18 The index values returned in indices by MPI_T_CATEGORY_GET_CVARS, 19MPI_T_CATEGORY_GET_PVARS and MPI_T_CATEGORY_GET_CATEGORIES can be used 20as input to MPI_T_CVAR_GET_INFO, MPI_T_PVAR_GET_INFO and 21MPI_T_CATEGORY_GET_INFO, respectively. 22 The user is responsible for allocating the arrays passed into the functions 23MPI_T_CATEGORY_GET_CVARS, MPI_T_CATEGORY_GET_PVARS and 24 MPI_T_CATEGORY_GET_CATEGORIES. Starting from array index 0, each function writes 2526up to len elements into the array. If the category contains more than len elements, the function returns an arbitrary subset of size len. Otherwise, the entire set of elements is 27returned in the beginning entries of the array, and any remaining array entries are not 28modified. 2930 Return Codes for the MPI Tool Information Interface 31 15.3.10 32 All functions defined as part of the MPI tool information interface return an integer error 33 code (see Table 15.7) to indicate whether the function was completed successfully or was 34 aborted. In the latter case the error code indicates the reason for not completing the routine. 35 Such errors neither impact the execution of the MPI process nor invoke MPI error handlers. 36 The MPI process continues executing regardless of the return code from the call. The MPI 37 implementation is not required to check all user-provided parameters; if a user passes invalid 38 parameter values to any routine the behavior of the implementation is undefined. 39 All error codes with the prefix MPI_T_ must be unique values and cannot overlap with 40 any other error codes or error classes returned by the MPI implementation. Further, they 41 shall be treated as MPI error classes as defined in Section 9.4 and follow the same rules and 42restrictions. In particular, they must satisfy: 43 44 $0 = MPI_SUCCESS < MPI_T_ERR_XXX \le MPI_ERR_LASTCODE.$ 4546Rationale. All MPI tool information interface functions must return error classes, be-47 cause applications cannot portably call MPI_ERROR_CLASS before MPI initialization 48 to map an arbitrary error code to an error class. (*End of rationale.*)

Unofficial Draft for Comment Only

15.3.11 Profiling Interface

All requirements for the profiling interfaces, as described in Section 15.2, also apply to the MPI tool information interface. All rules, guidelines, and recommendations from Section 15.2 apply equally to calls defined as part of the MPI tool information interface.

Return Code	Description
Return Codes for All Functions in the	he MPI Tool Information Interface
MPI_SUCCESS	Call completed successfully
MPI_T_ERR_INVALID	Invalid or bad parameter value(s)
MPI_T_ERR_MEMORY	Out of memory
MPI_T_ERR_NOT_INITIALIZED	Interface not initialized
MPI_T_ERR_CANNOT_INIT	Interface not in the state to be initialized
MPI_T_ERR_NOT_ACCESSIBLE	Requested functionality not accessible
Return Codes for Datatype Function	
MPI_T_ERR_INVALID_INDEX	The enumeration index is invalid
MPI_T_ERR_INVALID_ITEM	The item index queried is out of range
	(for MPI_T_ENUM_GET_ITEM only)
Baturn Codes for Variable Category	y, and Event Query Functions: MPI_T_*_GET_*
MPI_T_ERR_INVALID_INDEX	The variable or category index is invalid
MPI_T_ERR_INVALID_NAME	The variable of category name is invalid
Return Codes for Handle Functions:	(
MPI_T_ERR_INVALID_INDEX	The variable index is invalid
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
MPI_T_ERR_OUT_OF_HANDLES	No more handles available
Return Codes for Session Functions:	
MPI_T_ERR_OUT_OF_SESSIONS	No more sessions available
MPI_T_ERR_INVALID_SESSION	Session argument is not a valid session
Return Codes for Control Variable A	Access Functions: MPI_T_CVAR_{READ WRITE}
MPI_T_ERR_CVAR_SET_NOT_NOW	Variable cannot be set at this moment
MPI_T_ERR_CVAR_SET_NEVER	Variable cannot be set until end of execution
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
Return Codes for Performance Varia	able Access and Control:
MPI_T_PVAR_{START STOP READ	WRITE RESET READREST}
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
MPI_T_ERR_INVALID_SESSION	Session argument is not a valid session
MPI_T_ERR_PVAR_NO_STARTSTOP	Variable cannot be started or stopped (for
	MPI_T_PVAR_START and MPI_T_PVAR_STOP)
MPI_T_ERR_PVAR_NO_WRITE	Variable cannot be written or reset (for
	MPI_T_PVAR_WRITE and MPI_T_PVAR_RESET)
MPI_T_ERR_PVAR_NO_ATOMIC	Variable cannot be read and written atomically (for
	MPI_T_PVAR_READRESET)
Return Codes for Source Functions:	
MPI_T_ERR_INVALID_INDEX	The source index is invalid
MPI_T_ERR_NOT_SUPPORTED	Requested functionality not supported
Return Codes for Category Function	
MPI_T_ERR_INVALID_INDEX	The category index is invalid
	The caregory much is myanu
Table 15.7: Return codes used in	1 functions of the MPI tool information interface

Deprecated Interfaces

16.1 Deprecated since MPI-2.0

The following function is deprecated and is superseded by MPI_COMM_CREATE_KEYVAL in MPI-2.0. The language independent definition of the deprecated function is the same as that of the new function, except for the function name and a different behavior in the C/Fortran language interoperability, see Section 19.2.7. The language bindings are modified.

			21	
MPI_KE	YVAL_CREATE(copy_fr	n, delete_fn, keyval, extra_state)	22	
IN	copy_fn	Copy callback function for keyval	23	
IN	delete_fn		24	
IIN		Delete callback function for keyval	25	
OUT	keyval	key value for future access (integer)	26	
IN	extra_state	Extra state for callback functions	27	
			28	
C bindi	ing		29	
	0	Copy_function *copy_fn,	30	
		nction *delete_fn, int *keyval,	31	
	void *extra_s		32	
	VOIU *extia_s		33	
For this	routine, an interface wi	thin the mpi_f08 module was never defined.	34	
Fortron	binding		35	
	U		36	
	MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)			
	EXTERNAL COPY_FN, DELETE_FN INTEGER KEYVAL, EXTRA_STATE, IERROR			
LNI	EGER KEYVAL, EXIRA_	STATE, TERRUR	39	
The	copy_fn function is in	voked when a communicator is duplicated by	40	
MPI_CO	MM_DUP. copy_fn show	Ild be of type MPI_Copy_function, which is defined as follows:	41	
			42	
typedef	int MPI_Copy_funct	ion(MPI_Comm oldcomm, int keyval,	43	
7 1		tate, void *attribute_val_in,	44	
		te_val_out, int *flag);	45	
		C C	46	
		uch a function is as follows:	47	
For this	routine, an interface w	thin the mpi_f08 module was never defined.	48	

1 2 3 4 5 6	SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR) INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, IERR LOGICAL FLAG
7 8 9 10 11 12 13 14 15 16 17	<pre>copy_fn may be specified as MPI_NULL_COPY_FN or MPI_DUP_FN from either C or Fortran; MPI_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note that MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated. Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function, which is defined as follows: typedef int MPI_Delete_function(MPI_Comm comm, int keyval, void *attribute_val, void *extra_state);</pre>
18 19	A Fortran declaration for such a function is as follows: For this routine, an interface within the mpi_f08 module was never defined.
20 21 22	SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR) INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
23 24 25 26 27 28 29 30	delete_fn may be specified as MPI_NULL_DELETE_FN from either C or Fortran; MPI_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS. Note that MPI_NULL_DELETE_FN is also deprecated. The following function is deprecated and is superseded by MPI_COMM_FREE_KEYVAL in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.
31	MPI_KEYVAL_FREE(keyval)
32 33	INOUT keyval Frees the integer key value (integer)
34 35 36	C binding int MPI_Keyval_free(int *keyval)
37	For this routine, an interface within the mpi_f08 module was never defined.
38 39 40 41	Fortran binding MPI_KEYVAL_FREE(KEYVAL, IERROR) INTEGER KEYVAL, IERROR
42 43 44 45 46 47 48	The following function is deprecated and is superseded by MPI_COMM_SET_ATTR in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_ATTI	R_PUT(comm, keyval, attribute	e_val)	1
INOUT	comm	communicator to which attribute will be attached	2 3
		(handle)	4
IN	keyval	key value, as returned by MPI_KEYVAL_CREATE	5
		(integer)	6
IN	attribute_val	attribute value	7
			8
C bindin			9 10
int MPI_A	Attr_put(MPI_Comm comm, in	nt keyval, void *attribute_val)	10
For this re	outine, an interface within the	mpi_f08 module was never defined.	12
Fortran h	binding		13
	PUT(COMM, KEYVAL, ATTRIB	UTE_VAL, IERROR)	14
INTEC	ER COMM, KEYVAL, ATTRIBU	TE_VAL, IERROR	15
The f	ollowing function is deprecated	d and is superseded by MPI_COMM_GET_ATTR in	16 17
		inition of the deprecated function is the same as of	18
		name. The language bindings are modified.	19
	, 1		20
			21
MPI_ALTI	R_GET(comm, keyval, attribute	e_val, flag)	22
IN	comm	communicator to which attribute is attached (handle)	23
IN	keyval	key value (integer)	24
OUT	attribute_val	attribute value, unless $flag = false$	25
OUT	flag	true if an attribute value was extracted; false if no	26 27
001		attribute is associated with the key	27
			29
C bindin	g		30
		nt keyval, void *attribute_val, int *flag)	31
For this re	uting an interface within the	mpi_f08 module was never defined.	32
FOI UIIS IC	duffie, all'interface within the	mpi_100 module was never denned.	33
Fortran b			34
	GET(COMM, KEYVAL, ATTRIB		35
	ER COMM, KEYVAL, ATTRIBU	TE_VAL, IERRUR	36 37
LUGI(CAL FLAG		38
		and is superseded by $MPI_COMM_DELETE_ATTR$	39
		lefinition of the deprecated function is the same as	40
of the new	tunction, except of the funct	ion name. The language bindings are modified.	41
			42
MPI_ATTI	R_DELETE(comm, keyval)		43
INOUT	comm	communicator to which attribute is attached (handle)	44
			45
IN	keyval	The key value of the deleted attribute (integer)	46
C bindin	g		47 48

int MPI_Attr_delete(MPI_Comm comm, int keyval)

 $\mathbf{2}$ For this routine, an interface within the mpi_f08 module was never defined. 3

4Fortran binding

```
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
```

INTEGER COMM, KEYVAL, IERROR

7 8 9

10

13 14

15

16

17

18

19

202122

23

 24

252627

2829

30

 31 32

33

38

41

43

44

5

6

1

16.2Deprecated since MPI-2.2

11The entire set of C++ language bindings have been removed. See Chapter 17, Removed 12Interfaces for more information.

The following function typedefs have been deprecated and are superseded by new names. Other than the typedef names, the function signatures are exactly the same; the names were updated to match conventions of other function typedef names.

> Deprecated Name New Name MPI_Comm_errhandler_function MPI_Comm_errhandler_fn MPI_File_errhandler_function MPI_File_errhandler_fn MPI_Win_errhandler_fn MPI_Win_errhandler_function

16.3 Deprecated since MPI-3.2

Cancelling a send request by calling MPI_CANCEL has been deprecated and may be removed in a future version of the MPI specification.

Deprecated since MPI-4.0 16.4

The following function is deprecated and is superseded by the new MPI_INFO_GET_STRING call in MPI-4.0.

MPI_INFO_GET(info, key, valuelen, value, flag)

```
34
         IN
                     info
                                                        info object (handle)
35
         IN
                     key
                                                        key (string)
36
37
         IN
                     valuelen
                                                        length of value arg (integer)
         OUT
                     value
                                                        value (string)
39
         OUT
                     flag
                                                        true if key defined, false if not (boolean)
40
```

C binding 42

int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,

int *flag)

```
45
     Fortran 2008 binding
46
```

```
MPI_Info_get(info, key, valuelen, value, flag, ierror)
47
         TYPE(MPI_Info), INTENT(IN) :: info
48
```

	ACTER(LEN=*), INTENT(IN)	•	1
	GER, INTENT(IN) :: valuel		2 3
	ACTER(LEN=valuelen), INTE CAL, INTENT(OUT) :: flag	NT(UUT) :: Value	4
	GER, OPTIONAL, INTENT(OUT) :: ierror	5
			6
Fortran	DINGING _GET(INFO, KEY, VALUELEN,	VALUE FLAC TERROR)	7
	GER INFO, VALUELEN, IERRO		8
	ACTER*(*) KEY, VALUE		9 10
LOGI	CAL FLAG		11
The	following function is deprecat	ed and is superseded by the new	12
	D_GET_STRING call in MPI-4.		13
			14
MPI INFO	D_GET_VALUELEN(info, key, v	valuelen, flag)	15 16
IN	info	info object (handle)	17
			18
IN	key 	key (string)	19
OUT	valuelen	length of value arg (integer)	20
OUT	flag	true if key defined, false if not (boolean)	21 22
~			22
C bindin	0	a infationation that they int the lucion	24
int MP1_	int *flag)	o info, const char *key, int *valuelen,	25
T (C		26
	2008 binding _get_valuelen(info, key,	valuelen flag jerror)	27 28
	(MPI_Info), INTENT(IN) ::	-	28 29
	ACTER(LEN=*), INTENT(IN)		30
INTE	GER, INTENT(OUT) :: value	len	31
	CAL, INTENT(OUT) :: flag		32
INTE	GER, OPTIONAL, INTENT(OUT) :: ierror	33
Fortran	binding		34 35
	_GET_VALUELEN(INFO, KEY,		36
	GER INFO, VALUELEN, IERRO	R	37
	ACTER*(*) KEY CAL FLAG		38
			39
The	following return class has been	deprecated and is superseded by a new name.	40
	Deprecated N	ame Replacement Name	41 42
	MPI_T_ERR_INVALID_I	TEM MPI_T_ERR_INVALID_INDEX	43
The	following Fortran subroutines	are deprecated because the Fortran language	44
	_	ctions provide similar functionality. Note that while	45
-	0	ize in bytes, storage_size() provides the size in bits.	46
			47

```
1
      MPI_SIZEOF(x, size)
\mathbf{2}
        IN
                   Х
                                                  a Fortran variable of numeric intrinsic type (choice)
3
        OUT
                                                  size of machine representation of that type (integer)
                   size
4
\mathbf{5}
      Fortran 2008 binding
6
     MPI_Sizeof(x, size, ierror)
7
           TYPE(*), DIMENSION(..) :: x
8
           INTEGER, INTENT(OUT) :: size
9
10
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
      Fortran binding
12
     MPI_SIZEOF(X, SIZE, IERROR)
13
           <type> X
14
           INTEGER SIZE, IERROR
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

Removed Interfaces

17.1 Removed MPI-1 Bindings

17.1.1 Overview

The following MPI-1 bindings were deprecated as of MPI-2 and are removed in MPI-3. They may be provided by an implementation for backwards compatibility, but are not required. Removal of these bindings affects all language-specific definitions thereof. Only the language-neutral bindings are listed when possible.

17.1.2 Removed MPI-1 Functions

Table 17.1 shows the removed MPI-1 functions and their replacements.

Removed	MPI-2 Replacement
MPI_ADDRESS	MPI_GET_ADDRESS
MPI_ERRHANDLER_CREATE	MPI_COMM_CREATE_ERRHANDLER
MPI_ERRHANDLER_GET	MPI_COMM_GET_ERRHANDLER
MPI_ERRHANDLER_SET	MPI_COMM_SET_ERRHANDLER
MPI_TYPE_EXTENT	MPI_TYPE_GET_EXTENT
MPI_TYPE_HINDEXED	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_HVECTOR	MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_LB	MPI_TYPE_GET_EXTENT
MPI_TYPE_STRUCT	MPI_TYPE_CREATE_STRUCT
MPI_TYPE_UB	MPI_TYPE_GET_EXTENT

Table 17.1: Removed MPI-1 functions and their replacements

17.1.3 Removed MPI-1 Datatypes

Table 17.2 shows the removed MPI-1 datatypes and their replacements.

17.1.4 Removed MPI-1 Constants

Table 17.3 shows the removed MPI-1 constants. There are no MPI-2 replacements.

 $46 \\ 47$

	728		CHAPTER	17.	REMOVED INTERFACES
1	Т	Removed	MPI-2 Replacement		
2					
		MPI_LB	MPI_TYPE_CREATE_		
3	N	MPI_UB	MPI_TYPE_CREATE_	RES	IZED
4 5					
6	Table 17.2	: Remove	d MPI-1 datatypes and	their	r replacements
7		Re	moved MPI-1 Constant	s	
8		C type:	const int (or unnamed	enum)
9		Fortran	type: INTEGER		
10		MPI_CO	MBINER_HINDEXED_IN	TEGE	R
11		MPI_CO	MBINER_HVECTOR_INT	EGE	R
12		MPI_CO	MBINER_STRUCT_INTE	GER	
13					7
14		T-11-15	29. D		
15		Table 17	7.3: Removed MPI-1 co	nstar	its
16			_		
17 18	17.1.5 Removed MPI-1		51		
19	Table 17.4 shows the ren	noved MP	-1 callback prototypes	and	their MPI-2 replacements.
20 21	Reme	oved	MPI-2 Replacer	nent	
22	MPI_	Handler_fu	nction MPI_Comm_errh	andle	r_function
23					
24	Table 17 4. De	na arread MC	1 1 collibrate prototype	and	their perle comente
25	Table 17.4: Ref	moved IVIP	I-1 callback prototypes	and	their replacements
26					
27					
28	17.2 $C++$ Binding	<u>s</u>			
29	The C + + hindings were	denneset	ad as of MDL22. Th	a C	hinding one new ound in
30					++ bindings are removed in ngs may only be provided by
31	an implementation as de			man	igs may only be provided by
32	an implementation as de	scribed in	the MFI-2.2 Standard.		
33					
34					
35		The second se			
36					
37					
38					
39					
40					
41					
42					
43					
44					
45					
46					
47					
48					

Backward Incompatibilities

18.1 Backward Incompatible since MPI-3.2

MPI_COMM_DUP and MPI_COMM_IDUP no longer propagate info hints from the input communicator to the output communicator. This behavior can be achieved using MPI_COMM_DUP_WITH_INFO and MPI_COMM_IDUP_WITH_INFO.

The default communicator where errors are raised when not involving a communicator, window, or file was changed from MPI_COMM_WORLD to MPI_COMM_SELF.



Language Bindings

19.1 Fortran Support

19.1.1 Overview

The Fortran MPI language bindings have been designed to be compatible with the Fortran 90 standard with additional features from Fortran 2003 and Fortran 2008 [45] + TS 29113 [46].

Rationale. Fortran 90 contains numerous features designed to make it a more "modern" language than Fortran 77. It seems natural that MPI should be able to take advantage of these new features with a set of bindings tailored to Fortran 90. In Fortran 2008 + TS 29113, the major new language features used are the ASYNCHRONOUS attribute to protect nonblocking MPI operations, and assumed-type and assumed-rank dummy arguments for choice buffer arguments. Further requirements for compiler support are listed in Section 19.1.7. (*End of rationale.*)

MPI defines three methods of Fortran support:

- 1. USE mpi_f08: This method is described in Section 19.1.2. It requires compile-time argument checking with unique MPI handle types and provides techniques to fully solve the optimization problems with nonblocking calls. This is the only Fortran support method that is consistent with the Fortran standard (Fortran 2008 + TS 29113 and later). This method is highly recommended for all MPI applications.
- 2. USE mpi: This method is described in Section 19.1.3 and requires compile-time argument checking. Handles are defined as INTEGER. This Fortran support method is inconsistent with the Fortran standard, and its use is therefore not recommended. It exists only for backwards compatibility.
- 3. **INCLUDE 'mpif.h':** This method is described in Section 19.1.4. The use of the include file mpif.h is strongly discouraged starting with MPI-3.0, because this method neither guarantees compile-time argument checking nor provides sufficient techniques to solve the optimization problems with nonblocking calls, and is therefore inconsistent with the Fortran standard. It exists only for backwards compatibility with legacy MPI applications.

1 Compliant MPI-3 implementations providing a Fortran interface must provide one or $\mathbf{2}$ both of the following: 3 • The USE mpi_f08 Fortran support method. 4 5• The USE mpi and INCLUDE 'mpif.h' Fortran support methods. 6 $\overline{7}$ Section 19.1.6 describes restrictions if the compiler does not support all the needed features. 8 Application subroutines and functions may use either one of the modules or the mpif.h 9 include file. An implementation may require the use of one of the modules to prevent type 10mismatch errors. 11Advice to users. Users are advised to utilize one of the MPI modules even if mpif.h 12enforces type checking on a particular system. Using a module provides several poten-13 tial advantages over using an include file; the mpi_f08 module offers the most robust 14and complete Fortran support. (End of advice to users.) 1516In a single application, it must be possible to link together routines which USE mpi_f08. 17 USE mpi, and INCLUDE 'mpif.h'. 18 The LOGICAL compile-time constant MPI_SUBARRAYS_SUPPORTED is set to .TRUE. if 19all buffer choice arguments are defined in explicit interfaces with assumed-type and assumed-20rank [46]; otherwise it is set to .FALSE.. The LOGICAL compile-time constant 21MPI_ASYNC_PROTECTS_NONBLOCKING is set to .TRUE. if the ASYNCHRONOUS attribute was 22added to the choice buffer arguments of all nonblocking interfaces and the underlying 23Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of 24 TS 29113), otherwise it is set to .FALSE.. These constants exist for each Fortran support 25method, but not in the C header file. The values may be different for each Fortran support 26method. All other constants and the integer values of handles must be the same for each 27Fortran support method. 28Section 19.1.2 through 19.1.4 define the Fortran support methods. The Fortran in-29terfaces of each MPI routine are shorthands. Section 19.1.5 defines the corresponding 30 full interface specification together with the specific procedure names and implications for 31 the profiling interface. Section 19.1.6 the implementation of the MPI routines for differ-32 ent versions of the Fortran standard. Section 19.1.7 summarizes major requirements for 33 valid MPI-3.0 implementations with Fortran support. Section 19.1.8 and Section 19.1.9 de-34scribe additional functionality that is part of the Fortran support. MPI_F_SYNC_REG is 35 needed for one of the methods to prevent register optimization problems. A set of functions 36 provides additional support for Fortran intrinsic numeric types, including parameterized 37 types: MPI_TYPE_MATCH_SIZE, MPI_TYPE_CREATE_F90_INTEGER, 38 MPI_TYPE_CREATE_F90_REAL and MPI_TYPE_CREATE_F90_COMPLEX. In the context 39 of MPI, parameterized types are Fortran intrinsic types which are specified using KIND type 40 parameters. Sections 19.1.10 through 19.1.19 give an overview and details on known prob-41 lems when using Fortran together with MPI; Section 19.1.20 compares the Fortran problems 42with those in C. 43

43 44

19.1.2 Fortran Support Through the mpi_f08 Module

An MPI implementation providing a Fortran interface must provide a module named mpi_f08
 that can be used in a Fortran program. Section 19.1.6 describes restrictions if the compiler
 does not support all the needed features. Within all MPI function specifications, the first

Unofficial Draft for Comment Only

of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking for all arguments which are not TYPE(*), with the following exception:

Only one Fortran interface is defined for functions that are deprecated as of MPI-3.0. This interface must be provided as an explicit interface according to the rules defined for the mpi module, see Section 19.1.3.

Advice to users. It is strongly recommended that developers substitute calls to deprecated routines when upgrading from mpif.h or the mpi module to the mpi_f08 module. (End of advice to users.)

- Define the derived type MPI_Status, and define all MPI handles with uniquely named handle types (instead of INTEGER handles, as in the mpi module). This is reflected in the first Fortran binding in each MPI function definition throughout this document (except for the deprecated routines).
- Overload the operators .EQ. and .NE. to allow the comparison of these MPI handles with .EQ., .NE., == and /=.
- Use the ASYNCHRONOUS attribute to protect the buffers of nonblocking operations, and set the LOGICAL compile-time constant MPI_ASYNC_PROTECTS_NONBLOCKING to .TRUE. if the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TS 29113). See Section 19.1.6 for older compiler versions.

• Set the LOGICAL compile-time constant MPI_SUBARRAYS_SUPPORTED to .TRUE. and declare choice buffers using the Fortran 2008 TS 29113 features assumed-type and assumed-rank, i.e., TYPE(*), DIMENSION(..) in all nonblocking, split collective and persistent communication routines, if the underlying Fortran compiler supports it. With this, non-contiguous sub-arrays can be used as buffers in nonblocking routines.

Rationale. In all blocking routines, i.e., if the choice-buffer is not declared as ASYNCHRONOUS, the TS 29113 feature is not needed for the support of noncontiguous buffers because the compiler can pass the buffer by in-and-out-copy through a contiguous scratch array. (*End of rationale.*)

- Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the Fortran 2008 TS 29113 assumed-type and assumed-rank notation. In this case, the use of non-contiguous sub-arrays as buffers in nonblocking calls may be invalid. See Section 19.1.6 for details.
- Declare each argument with an INTENT of IN, OUT, or INOUT as defined in this standard.

 $\mathbf{2}$

 $\frac{4}{5}$

 31

Rationale. For these definitions in the mpi_f08 bindings, in most cases, INTENT(IN) is used if the C interface uses call-by-value. For all buffer arguments and for OUT and INOUT dummy arguments that allow one of the non-ordinary Fortran constants (see MPI_BOTTOM, etc. in Section 2.5.4) as input, an INTENT is not specified. (End of rationale.)

Advice to users. If a dummy argument is declared with INTENT(OUT), then the Fortran standard stipulates that the actual argument becomes undefined upon invocation of the MPI routine, i.e., it may be overwritten by some other values, e.g. zeros; according to [45], 12.5.2.4 Ordinary dummy variables, Paragraph 17: "If a dummy argument has INTENT(OUT), the actual argument becomes undefined at the time the association is established, except [...]". For example, if the dummy argument is an assumed-size array and the actual argument is a strided array, the call may be implemented with copy-in and copy-out of the argument. In the case of INTENT(OUT) the copy-in may be suppressed by the optimization and the routine starts execution using an array of undefined values. If the routine stores fewer elements into the dummy argument than is provided in the actual argument, then the remaining locations are overwritten with these undefined values. See also both advices to implementors in Section 19.1.3. (End of advice to users.)

• Declare all ierror output arguments as OPTIONAL, except for user-defined callback functions (e.g., COMM_COPY_ATTR_FUNCTION) and predefined callbacks (e.g., MPI_COMM_NULL_COPY_FN).

Rationale. For user-defined callback functions (e.g., COMM_COPY_ATTR_FUNCTION) and their predefined callbacks (e.g., MPI_COMM_NULL_COPY_FN), the ierror argument is not optional. The MPI library must always call these routines with an actual ierror argument. Therefore, these user-defined functions need not check whether the MPI library calls these routines with or without an actual ierror output argument. (*End of rationale.*)

The MPI Fortran bindings in the mpi_f08 module are designed based on the Fortran 2008 standard [45] together with the Technical Specification "TS 29113 Further Interoperability with C" [46] of the ISO/IEC JTC1/SC22/WG5 (Fortran) working group.

Rationale. The features in TS 29113 on further interoperability with C were decided on by ISO/IEC JTC1/SC22/WG5 and designed by PL22.3 (formerly J3) to support a higher level of integration between Fortran-specific features and C than was provided in the Fortran 2008 standard; part of this design is based on requirements from the MPI Forum to support MPI-3.0. According to [46], "an ISO/IEC TS is reviewed after three years in order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn. If the ISO/IEC TS is confirmed, it is reviewed again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn."

The TS 29113 contains the following language features that are needed for the MPI bindings in the mpi_f08 module: assumed-type and assumed-rank. It is important that any possible actual argument can be used for such dummy arguments, e.g., scalars, arrays, assumed-shape arrays, assumed-size arrays, allocatable arrays, and

with any element type, e.g., REAL, CHARACTER*5, CHARACTER*(*), sequence derived types, or BIND(C) derived types. Especially for backward compatibility reasons, it is important that any possible actual argument in an implicit interface implementation of a choice buffer dummy argument (e.g., with mpif.h without argument-checking) can be used in an implementation with assumed-type and assumed-rank argument in an explicit interface (e.g., with the mpi_f08 module).

A further feature useful for MPI is the extension of the semantics of the ASYNCHRONOUS attribute: In F2003 and F2008, this attribute could be used only to protect buffers of Fortran asynchronous I/O. With TS 29113, this attribute now also covers asynchronous communication occurring within library routines written in C.

The MPI Forum hereby wishes to acknowledge this important effort by the Fortran PL22.3 and WG5 committee. (*End of rationale.*)

19.1.3 Fortran Support Through the mpi Module

An MPI implementation providing a Fortran interface must provide a module named mpi that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking and allows positional and keyword-based argument lists. If an implementation is paired with a compiler that either does not support TYPE(*), DIMENSION(..) from TS 29113, or is otherwise unable to ignore the types of choice buffers, then the implementation must provide explicit interfaces only for MPI routines with no choice buffer arguments. See Section 19.1.6 for more details.
- Define all MPI handles as type INTEGER.
- Define the derived type MPI_Status and all named handle types that are used in the mpi_f08 module. For these named handle types, overload the operators .EQ. and .NE. to allow handle comparison via the .EQ., .NE., == and /= operators.

Rationale. They are needed only when the application converts old-style INTEGER handles into new-style handles with a named type. (*End of rationale.*)

- A high quality MPI implementation may enhance the interface by using the ASYNCHRONOUS attribute in the same way as in the mpi_f08 module if it is supported by the underlying compiler.
- Set the LOGICAL compile-time constant MPI_ASYNC_PROTECTS_NONBLOCKING to .TRUE. if the ASYNCHRONOUS attribute is used in all nonblocking interfaces and the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TS 29113), otherwise to .FALSE..

Unofficial Draft for Comment Only

 $\mathbf{2}$

For an MPI implementation that fully supports nonblocking calls Advice to users. with the ASYNCHRONOUS attribute for choice buffers, an existing MPI-2.2 application may fail to compile even if it compiled and executed with expected results with an MPI-2.2 implementation. One reason may be that the application uses "contiguous" but not "simply contiguous" ASYNCHRONOUS arrays as actual arguments for choice buffers of nonblocking routines, e.g., by using subscript triplets with stride one or specifying (1:n) for a whole dimension instead of using (:). This should be fixed to fulfill the Fortran constraints for ASYNCHRONOUS dummy arguments. This is not considered a violation of backward compatibility because existing applications can not use the ASYNCHRONOUS attribute to protect nonblocking calls. Another reason may be that the application does not conform either to MPI-2.2, or to MPI-3.0, or to the Fortran standard, typically because the program forces the compiler to perform copy-in/out for a choice buffer argument in a nonblocking MPI call. This is also not a violation of backward compatibility because the application itself is non-conforming. See Section 19.1.12 for more details. (End of advice to users.)

- A high quality MPI implementation may enhance the interface by using TYPE(*), DIMENSION(..) choice buffer dummy arguments instead of using non-standardized extensions such as !\$PRAGMA IGNORE_TKR or a set of overloaded functions as described by M. Hennecke in [32], if the compiler supports this TS 29113 language feature. See Section 19.1.6 for further details.
 - Set the LOGICAL compile-time constant MPI_SUBARRAYS_SUPPORTED to .TRUE. if all choice buffer arguments in all nonblocking, split collective and persistent communication routines are declared with TYPE(*), DIMENSION(..), otherwise set it to .FALSE.. When MPI_SUBARRAYS_SUPPORTED is defined as .TRUE., non-contiguous sub-arrays can be used as buffers in nonblocking routines.
 - Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the TS 29113 assumed-type and assumed-rank features. In this case, the use of non-contiguous sub-arrays in non-blocking calls may be disallowed. See Section 19.1.6 for details.

An MPI implementation may provide other features in the mpi module that enhance the usability of MPI while maintaining adherence to the standard. For example, it may provide INTENT information in these interface blocks.

Advice to implementors. The appropriate INTENT may be different from what is given in the MPI language-neutral bindings. Implementations must choose INTENT so that the function adheres to the MPI standard, e.g., by defining the INTENT as provided in the mpi_f08 bindings. (End of advice to implementors.)

Rationale. The intent given by the MPI generic interface is not precisely defined
 and does not in all cases correspond to the correct Fortran INTENT. For instance,
 receiving into a buffer specified by a datatype with absolute addresses may require
 associating MPI_BOTTOM with a dummy OUT argument. Moreover, "constants" such
 MPI_BOTTOM and MPI_STATUS_IGNORE are not constants as defined by Fortran,
 but "special addresses" used in a nonstandard way. Finally, the MPI-1 generic intent

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21 22

23

24

25

26

27 28

29

30

31

32

33 34

35

36

37 38

39

40

41

was changed in several places in MPI-2. For instance, MPI_IN_PLACE changes the intent of an OUT argument to be INOUT. (*End of rationale.*)

Advice to implementors. The Fortran 2008 standard illustrates in its Note 5.17 that "INTENT(OUT) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If an argument should retain its value rather than being redefined, INTENT(INOUT) should be used rather than INTENT(OUT), even if there is no explicit reference to the value of the dummy argument. Furthermore, INTENT(INOUT) is not equivalent to omitting the IN-TENT attribute, because INTENT(INOUT) always requires that the associated actual argument is definable." Applications that include mpif.h may not expect that INTENT(OUT) is used. In particular, output array arguments are expected to keep their content as long as the MPI routine does not modify them. To keep this behavior, it is recommended that implementations not use INTENT(OUT) in the mpi module and the mpif.h include file, even though INTENT(OUT) is specified in an interface description of the mpi_f08 module. (End of advice to implementors.)

19.1.4 Fortran Support Through the mpif.h Include File

The use of the mpif.h include file is strongly discouraged and may be deprecated in a future version of MPI.

An MPI implementation providing a Fortran interface must provide an include file named mpif.h that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is supported by this include file. This include file must:

• Define all named MPI constants. • Declare MPI functions that return a value. • Define all handles as INTEGER. • Be valid and equivalent for both fixed and free source form. For each MPI routine, an implementation can choose to use an implicit or explicit interface for the second Fortran binding (in deprecated routines, the first one may be omitted). • Set the LOGICAL compile-time constants MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING according to the same rules as for the mpi module. In the case of implicit interfaces for choice buffer or nonblocking routines, the constants must be set to .FALSE.. Advice to users. Instead of using mpif.h, the use of the mpi_f08 or mpi module is strongly encouraged for the following reasons: • Most mpif.h implementations do not include compile-time argument checking. • Therefore, many bugs in MPI applications remain undetected at compile-time, such as: - Missing ierror as last argument in most Fortran bindings.

Unofficial Draft for Comment Only

 $\overline{7}$

 $46 \\ 47$

	738 CHAPTER 19. LANGUAGE BINDINGS
1 2	 Declaration of a status as an INTEGER variable instead of an INTEGER array with size MPI_STATUS_SIZE.
$\frac{3}{4}$	- Incorrect argument positions; e.g., interchanging the count and
5	datatype arguments. – Passing incorrect MPI handles; e.g., passing a datatype instead of a commu-
6 7	nicator.
8	• The migration from mpif.h to the mpi module should be relatively straightfor-
9 10	ward (i.e., substituting include 'mpif.h' after an implicit statement by use mpi before that implicit statement) as long as the application syntax is correct.
11	• Migrating portable and correctly written applications to the mpi module is not
12 13	expected to be difficult. No compile or runtime problems should occur because an mpif.h include file was always allowed to provide explicit Fortran interfaces.
14 15	(End of advice to users.)
16 17	Rationale. With MPI-3.0, the mpif.h include file was not deprecated in order to
18	retain strong backward compatibility. Internally, mpif.h and the mpi module may be implemented so that essentially the same library implementation of the MPI routines
19 20	can be used. (<i>End of rationale.</i>)
21	10.1 E. Jata G
22	19.1.5 Interface Specifications, Procedure Names, and the Profiling Interface
23 24	The Fortran interface specification of each MPI routine specifies the routine name that must be called by the application program, and the names and types of the dummy arguments
25	together with additional attributes. The Fortran standard allows a given Fortran interface
26 27	to be implemented with several methods, e.g., within or outside of a module, with or without BIND(C), or the buffers with or without TS 29113. Such implementation decisions imply
28	different binary interfaces and different specific procedure names. The requirements for
29	several implementation schemes together with the rules for the specific procedure names
30 31	and its implications for the profiling interface are specified within this section, but not the implementation details.
32	
33 34	<i>Rationale.</i> This section was introduced in MPI-3.0 on Sep. 21, 2012. The major goals for implementing the three Fortran support methods have been:
35 36 37	• Portable implementation of the wrappers from the MPI Fortran interfaces to the MPI routines in C.
38	• Binary backward compatible implementation path when switching
39	MPI_SUBARRAYS_SUPPORTED from .FALSE. to .TRUE
40 41	• The Fortran PMPI interface need not be backward compatible, but a method must be included that a tools layer can use to examine the MPI library about
42	the specific procedure names and interfaces used.
$43 \\ 44$	• No performance drawbacks.
45	• Consistency between all three Fortran support methods.
46	• Consistent with Fortran $2008 + TS 29113$.
47 48	

No.	Specific pro- cedure name	Calling convention
1A	MPI_Isend_f08	Fortran interface and arguments, as in Annex A.4, except
		that in routines with a choice buffer dummy argument, this
		dummy argument is implemented with non-standard ex-
		tensions like ! \$PRAGMA IGNORE_TKR, which provides a call-
		by-reference argument without type, kind, and dimension
		checking.
В	MPI_Isend_f08ts	Fortran interface and arguments, as in Annex A.4, but
		only for routines with one or more choice buffer dummy
		arguments; these dummy arguments are implemented with
		TYPE(*), DIMENSION().
4	MPI_ISEND	Fortran interface and arguments, as in Annex A.5, except
		that in routines with a choice buffer dummy argument, this
		dummy argument is implemented with non-standard ex-
		tensions like !\$PRAGMA IGNORE_TKR, which provides a call-
		by-reference argument without type, kind, and dimension
		checking.
В	MPI_ISEND_FTS	Fortran interface and arguments, as in Annex A.5, but
		only for routines with one or more choice buffer dummy
		arguments; these dummy arguments are implemented with
		TYPE(*), DIMENSION().

Table 19.1: Specific Fortran procedure names and related calling conventions. MPI_ISEND is used as an example. For routines without choice buffers, only 1A and 2A apply.

The design expected that all dummy arguments in the MPI Fortran interfaces are interoperable with C according to Fortran 2008 + TS 29113. This expectation was not fulfilled. The LOGICAL arguments are not interoperable with C, mainly because the internal representations for .FALSE. and .TRUE. are compiler dependent. The provided interface was mainly based on BIND(C) interfaces and therefore inconsistent with Fortran. To be consistent with Fortran, the BIND(C) had to be removed from the callback procedure interfaces and the predefined callbacks, e.g., MPI_COMM_DUP_FN. Non-BIND(C) procedures are also not interoperable with C, and therefore the BIND(C) had to be removed from all routines with PROCEDURE arguments, e.g., from MPI_OP_CREATE.

Therefore, this section was rewritten as an erratum to MPI-3.0. (End of rationale.)

A Fortran call to an MPI routine shall result in a call to a procedure with one of the specific procedure names and calling conventions, as described in Table 19.1. Case is not significant in the names.

Note that for the deprecated routines in Section 16.1, which are reported only in Annex A.5, scheme 2A is utilized in the mpi module and mpif.h, and also in the mpi_f08 module.

To set MPI_SUBARRAYS_SUPPORTED to .TRUE. within a Fortran support method, it is required that all nonblocking and split-collective routines with buffer arguments are implemented according to 1B and 2B, i.e., with MPI_Xxxx_f08ts in the mpi_f08 module,

Unofficial Draft for Comment Only

and with MPI_XXXX_FTS in the mpi module and the mpif.h include file.

The mpi and mpi_f08 modules and the mpif.h include file will each correspond to exactly one implementation scheme from Table 19.1. However, the MPI library may contain multiple implementation schemes from Table 19.1.

Advice to implementors. This may be desirable for backwards binary compatibility in the scope of a single MPI implementation, for example. (*End of advice to implementors.*)

9 Rationale. After a compiler provides the facilities from TS 29113, i.e., TYPE(*), 10 DIMENSION(...), it is possible to change the bindings within a Fortran support method 11 to support subarrays without recompiling the complete application provided that the 12previous interfaces with their specific procedure names are still included in the li-13 brary. Of course, only recompiled routines can benefit from the added facilities. 14There is no binary compatibility conflict because each interface uses its own spe-15cific procedure names and all interfaces use the same constants (except the value of 16MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING) and type 17 definitions. After a compiler also ensures that buffer arguments of nonblocking MPI 18 operations can be protected through the ASYNCHRONOUS attribute, and the proce-19 dure declarations in the mpi_f08 and mpi module and the mpif.h include file declare 20choice buffers with the ASYNCHRONOUS attribute, then the value of

²¹ MPI_ASYNC_PROTECTS_NONBLOCKING can be switched to .TRUE. in the module def-²² inition and include file. (*End of rationale.*)

Advice to users. Partial recompilation of user applications when upgrading MPI implementations is a highly complex and subtle topic. Users are strongly advised to consult their MPI implementation's documentation to see exactly what is — and what is not — supported. (End of advice to users.)

Within the mpi_f08 and mpi modules and mpif.h, for all MPI procedures, a second procedure with the same calling conventions shall be supplied, except that the name is modified by prefixing with the letter "P", e.g., PMPI_lsend. The specific procedure names for these PMPI_Xxxx procedures must be different from the specific procedure names for the MPI_Xxxx procedures and are not specified by this standard.

A user-written or middleware profiling routine should provide the same specific Fortran procedure names and calling conventions, and therefore can interpose itself as the MPI library routine. The profiling routine can internally call the matching

PMPI routine with any of its existing bindings, except for routines that have callback routine dummy arguments, choice buffer arguments, or that are attribute caching routines (

³⁵ MPI_{COMM|WIN|TYPE}_{SET|GET}_ATTR). In this case, the profiling software should ⁴⁰ invoke the corresponding PMPI routine using the same Fortran support method as used in ⁴¹ the calling application program, because the C, mpi_f08 and mpi callback prototypes are ⁴² different or the meaning of the choice buffer or attribute_val arguments are different.

Advice to users. Although for each support method and MPI routine (e.g.,

⁴⁴ MPI_ISEND in mpi_f08), multiple routines may need to be provided to intercept ⁴⁵ the specific procedures in the MPI library (e.g., MPI_Isend_f08 and MPI_Isend_f08ts), ⁴⁶ each profiling routine itself uses only one support method (e.g., mpi_f08) and calls ⁴⁷ the real MPI routine through the one PMPI routine defined in this support method ⁴⁸ (i.e., PMPI_Isend in this example). (*End of advice to users.*)

Unofficial Draft for Comment Only

1

5

6

7

8

Advice to implementors. If all of the following conditions are fulfilled:

- the handles in the mpi_f08 module occupy one Fortran numerical storage unit (same as an INTEGER handle),
- the internal argument passing mechanism used to pass an actual ierror argument to a non-optional ierror dummy argument is binary compatible to passing an actual ierror argument to an ierror dummy argument that is declared as OPTIONAL,
- the internal argument passing mechanism for ASYNCHRONOUS and non-ASYNCHRONOUS arguments is the same,
- the internal routine call mechanism is the same for the Fortran and the C compilers for which the MPI library is compiled,
- the compiler does not provide TS 29113,

then the implementor may use the same internal routine implementations for all Fortran support methods but with several different specific procedure names. If the accompanying Fortran compiler supports TS 29113, then the new routines are needed only for routines with choice buffer arguments. (*End of advice to implementors.*)

Advice to implementors. In the Fortran support method mpif.h, compile-time argument checking can be also implemented for all routines. For mpif.h, the argument names are not specified through the MPI standard, i.e., only positional argument lists are defined, and not key-word based lists. Due to the rule that mpif.h must be valid for fixed and free source form, the subroutine declaration is restricted to one line with 72 characters. To keep the argument lists short, each argument name can be shortened to a minimum of one character. With this, the two longest subroutine declaration statements are

SUBROUTINE PMPI_Dist_graph_create_adjacent(a,b,c,d,e,f,g,h,i,j,k)
SUBROUTINE PMPI_Rget_accumulate(a,b,c,d,e,f,g,h,i,j,k,l,m,n)

with 71 and 66 characters. With buffers implemented with TS 29113, the specific procedure names have an additional postfix. The longest of such interface definitions is

INTERFACE PMPI_Rget_accumulate
SUBROUTINE PMPI_Rget_accumulate_fts(a,b,c,d,e,f,g,h,i,j,k,l,m,n)

with 70 characters. In principle, continuation lines would be possible in mpif.h (spaces in columns 73–131, & in column 132, and in column 6 of the continuation line) but this would not be valid if the source line length is extended with a compiler flag to 132 characters. Column 133 is also not available for the continuation character because lines longer than 132 characters are invalid with some compilers by default.

The longest specific procedure names are PMPI_Dist_graph_create_adjacent_f08 and PMPI_File_write_ordered_begin_f08ts both with 35 characters in the mpi_f08 module.

For example, the interface specifications together with the specific procedure names can be implemented with

 24

```
1
           MODULE mpi_f08
2
             TYPE, BIND(C) :: MPI_Comm
3
               INTEGER :: MPI_VAL
             END TYPE MPI_Comm
4
             . . .
5
             INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
6
               SUBROUTINE MPI_Comm_rank_f08(comm, rank, ierror)
7
                 IMPORT :: MPI_Comm
8
                 TYPE(MPI_Comm),
                                       INTENT(IN) :: comm
9
                                       INTENT(OUT) :: rank
                 INTEGER,
                 INTEGER, OPTIONAL,
                                       INTENT(OUT) :: ierror
10
               END SUBROUTINE
11
             END INTERFACE
12
           END MODULE mpi_f08
13
14
           MODULE mpi
15
             INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
16
               SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
                                                  ! The INTENT may be added although
17
                 INTEGER, INTENT(IN) :: comm
                 INTEGER, INTENT(OUT) :: rank
                                                  ! it is not defined in the
18
                 INTEGER, INTENT(OUT) :: ierror ! official routine definition.
19
               END SUBROUTINE
20
             END INTERFACE
21
           END MODULE mpi
22
23
           And if interfaces are provided in mpif.h, they might look like this (outside of any
24
           module and in fixed source format):
25
           !23456789012345678901234567890123456789012345678901234567890123456789012
26
                 INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
27
                  SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
28
                   INTEGER, INTENT(IN) :: comm ! The argument names may be
29
                   INTEGER, INTENT(OUT) :: rank
                                                    ! shortened so that the
30
                   INTEGER, INTENT(OUT) :: ierror ! subroutine line fits to the
31
                  END SUBROUTINE
                                                    ! maximum of 72 characters.
32
                 END INTERFACE
33
34
           (End of advice to implementors.)
35
           Advice to users.
                             The following is an example of how a user-written or middleware
36
           profiling routine can be implemented:
37
38
           SUBROUTINE MPI_Isend_f08ts(buf,count,datatype,dest,tag,comm,request,ierror)
39
             USE :: mpi_f08, my_noname => MPI_Isend_f08ts
40
             TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
41
             INTEGER,
                                  INTENT(IN)
                                                    :: count, dest, tag
42
             TYPE(MPI_Datatype), INTENT(IN)
                                                    :: datatype
             TYPE(MPI_Comm),
                                  INTENT(IN)
                                                    :: comm
43
             TYPE(MPI_Request), INTENT(OUT)
                                                    :: request
44
             INTEGER, OPTIONAL,
                                  INTENT(OUT)
                                                    :: ierror
45
               ! ... some code for the begin of profiling
46
             call PMPI_Isend (buf, count, datatype, dest, tag, comm, request, ierror)
47
               ! ... some code for the end of profiling
48
           END SUBROUTINE MPI_Isend_f08ts
```

Unofficial Draft for Comment Only

Note that this routine is used to intercept the existing specific procedure name MPI_lsend_f08ts in the MPI library. This routine must not be part of a module. This routine itself calls PMPI_lsend. The USE of the mpi_f08 module is needed for definitions of handle types and the interface for PMPI_lsend. However, this module also contains an interface definition for the specific procedure name MPI_lsend_f08ts that conflicts with the definition of this profiling routine (i.e., the name is doubly defined). Therefore, the USE here specifically excludes the interface from the module by renaming the unused routine name in the mpi_f08 module into "my_noname" in the scope of this routine. (*End of advice to users.*)

The PMPI interface allows intercepting MPI routines. For exam-Advice to users. 11 ple, an additional MPI_ISEND profiling wrapper can be provided that is called by the 12application and internally calls PMPI_ISEND. There are two typical use cases: a pro-13 filing layer that is developed independently from the application and the MPI library, 14and profiling routines that are part of the application and have access to the appli-15cation data. With MPI-3.0, new Fortran interfaces and implementation schemes were 16 introduced that have several implications on how Fortran MPI routines are internally 17 implemented and optimized. For profiling layers, these schemes imply that several in-18 ternal interfaces with different specific procedure names may need to be intercepted, 19 as shown in the example code above. Therefore, for wrapper routines that are part 20of a Fortran application, it may be more convenient to make the name shift within 21the application, i.e., to substitute the call to the MPI routine (e.g., MPI_ISEND) by a 22call to a user-written profiling wrapper with a new name (e.g., X_MPI_ISEND) and to 23call the Fortran MPI_ISEND from this wrapper, instead of using the PMPI interface. 24 (End of advice to users.) 25

Advice to implementors. An implementation that provides a Fortran interface must provide a combination of MPI library and module or include file that uses the specific procedure names as described in Table 19.1 so that the MPI Fortran routines are interceptable as described above. (*End of advice to implementors.*)

19.1.6 MPI for Different Fortran Standard Versions

This section describes which Fortran interface functionality can be provided for different versions of the Fortran standard.

- For Fortran 77 with some extensions:
 - MPI identifiers may be up to 30 characters (31 with the profiling interface).
 - MPI identifiers may contain underscores after the first character.
 - An MPI subroutine with a choice argument may be called with different argument types.
 - Although not required by the MPI standard, the INCLUDE statement should be available for including mpif.h into the user application source code.

Only MPI-1.1, MPI-1.2, and MPI-1.3 can be implemented. The use of absolute addresses from MPI_ADDRESS and MPI_BOTTOM may cause problems if an address does not fit into the memory space provided by an INTEGER. (In MPI-2.0 this problem is solved with MPI_GET_ADDRESS, but not for Fortran 77.)
45
46
47
48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8 9

10

26

27

28

29

30 31

32 33

34

35

36 37

38

39 40

41

42

43

1	• For Fortran 90:
2	The major additional features that are needed from Fortran 90 are:
3	
4	- The MODULE and INTERFACE concept.
5 6	- The KIND= and SELECTEDKIND concept.
7	- Fortran derived TYPEs and the SEQUENCE attribute.
8	- The OPTIONAL attribute for dummy arguments.
9	- Cray pointers, which are a non-standard compiler extension, are needed for the
10	use of MPI_ALLOC_MEM.
11	With these features, MPI-1.1 – MPI-2.2 can be implemented without restrictions.
12	MPI-3.0 can be implemented with some restrictions. The Fortran support methods
13 14	are abbreviated with $S1 = \text{the mpi}_f08 \text{ module}$, $S2 = \text{the mpi} \text{ module}$, and $S3 = \text{the}$
15	mpif.f include file. If not stated otherwise, restrictions exist for each method which
16	prevent implementing the complete semantics of MPI-3.0.
17	- MPI_SUBARRAYS_SUPPORTED equals .FALSE., i.e., subscript triplets and non-
18	contiguous subarrays cannot be used as buffers in nonblocking routines, RMA,
19	or split-collective I/O.
20 21	- S1, S2, and S3 can be implemented, but for S1, only a preliminary implementa-
21	tion is possible.
23	- In this preliminary interface of S1, the following changes are necessary:
24	* TYPE(*), DIMENSION() is substituted by non-standardized extensions
25	like !\$PRAGMA IGNORE_TKR.
26	* The ASYNCHRONOUS attribute is omitted.
27 28	* PROCEDURE() callback declarations are substituted by EXTERNAL .
29	- The specific procedure names are specified in Section 19.1.5.
30	– Due to the rules specified in Section 19.1.5, choice buffer declarations should be
31	implemented only with non-standardized extensions like !\$PRAGMA IGNORE_TKR
32	(as long as $F2008+TS$ 29113 is not available).
33 34	In S2 and S3: Without such extensions, routines with choice buffers should be
35	provided with an implicit interface, instead of overloading with a different MPI
36	function for each possible buffer type (as mentioned in Section 19.1.11). Such
37	overloading would also imply restrictions for passing Fortran derived types as
38	choice buffer, see also Section 19.1.15.
39	Only in S1: The implicit interfaces for routines with choice buffer arguments imply that the ierror argument cannot be defined as OPTIONAL. For this reason,
40	it is recommended not to provide the mpi_f08 module if such an extension is not
41 42	available.
42	- The ASYNCHRONOUS attribute can not be used in applications to protect buffers
44	in nonblocking MPI calls ($S1-S3$).
45	- The TYPE(C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE
46	routines is not available.
47	
48	

- In S1 and S2, the definition of the handle types (e.g., TYPE(MPI_Comm) and the status type TYPE(MPI_Status) must be modified: The SEQUENCE attribute 3 must be used instead of BIND(C) (which is not available in Fortran 90/95). This 4 restriction implies that the application must be fully recompiled if one switches to an MPI library for Fortran 2003 and later because the internal memory size of the handles may have changed. For this reason, an implementor may choose not to provide the mpi_f08 module for Fortran 90 compilers. In this case, the mpi_f08 handle types and all routines, constants and types related to TYPE(MPI_Status) (see Section 19.2.5) are also not available in the mpi module and mpif.h.
- For Fortran 95:

The quality of the MPI interface and the restrictions are the same as with Fortran 90.

• For Fortran 2003:

The major features that are needed from Fortran 2003 are:

- Interoperability with C, i.e.,
 - * BIND(C) derived types.
 - * The ISO_C_BINDING intrinsic type C_PTR and routine C_F_POINTER.
- The ability to define an ABSTRACT INTERFACE and to use it for PROCEDURE dummy arguments.
- The ability to overload the operators .EQ. and .NE. to allow the comparison of derived types (used in MPI-3.0 for MPI handles).
- The ASYNCHRONOUS attribute is available to protect Fortran asynchronous I/O. This feature is not vet used by MPI, but it is the basis for the enhancement for MPI communication in the TS 29113.

With these features (but still without the features of TS 29113), MPI-1.1 – MPI-2.2 can be implemented without restrictions, but with one enhancement:

- The user application can use TYPE(C_PTR) together with MPI_ALLOC_MEM as long as MPI_ALLOC_MEM is defined with an implicit interface because a C_PTR and an INTEGER(KIND=MPI_ADDRESS_KIND) argument must both map to a void * argument.

MPI-3.0 can be implemented with the following restrictions:

- MPI_SUBARRAYS_SUPPORTED equals .FALSE..
- For S1, only a preliminary implementation is possible. The following changes are necessary:
 - * TYPE(*), DIMENSION(..) is substituted by non-standardized extensions like !\$PRAGMA IGNORE_TKR.
- The specific procedure names are specified in Section 19.1.5.
- With S1, the ASYNCHRONOUS is required as specified in the second Fortran interfaces. With S2 and S3 the implementation can also add this attribute if explicit interfaces are used.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

5

6 $\overline{7}$

8

9 10

11

1213

14

1516

17

18

19

20

2122

23

 24

25

26

2728

29

30

31

32

33

34 35

36 37

38

39

40

41

42

43 44

45

46

1 2 3 4	 The ASYNCHRONOUS Fortran attribute can be used in applications to <i>try to</i> protect buffers in nonblocking MPI calls, but the protection can work only if the compiler is able to protect asynchronous Fortran I/O and makes no difference between such asynchronous Fortran I/O and MPI communication.
5 6 7	 The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can be used only for Fortran types that are C compatible.
8 9 10	 The same restriction as for Fortran 90 applies if non-standardized extensions like PRAGMA IGNORE_TKR are not available.
11 12 13	• For Fortran 2008 + TS 29113 and later and For Fortran 2003 + TS 29113: The major feature that are needed from TS 29113 are:
14 15 16 17 18 19	 TYPE(*), DIMENSION() is available. The ASYNCHRONOUS attribute is extended to protect also nonblocking MPI communication. The array dummy argument of the ISO_C_BINDING intrinsic C_F_POINTER is not restricted to Fortran types for which a corresponding type in C exists.
20 21	Using these features, MPI-3.0 can be implemented without any restrictions.
22 23 24 25 26	 With S1, MPI_SUBARRAYS_SUPPORTED equals .TRUE The ASYNCHRONOUS attribute can be used to protect buffers in nonblocking MPI calls. The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can be used for any Fortran type.
27 28 29 30	 With S2 and S3, the value of MPI_SUBARRAYS_SUPPORTED is implementation dependent. A high quality implementation will also provide MPI_SUBARRAYS_SUPPORTED==.TRUE. and will use the ASYNCHRONOUS attribute in the same way as in S1.
31 32 33	 If non-standardized extensions like !\$PRAGMA IGNORE_TKR are not available then S2 must be implemented with TYPE(*), DIMENSION().
34 35 36	Advice to implementors. If MPI_SUBARRAYS_SUPPORTED==.FALSE., the choice argument may be implemented with an explicit interface using compiler directives, for example:
37 38 39	INTERFACE SUBROUTINE MPI(buf,)
40 41 42	!DEC\$ ATTRIBUTES NO_ARG_CHECK :: buf !\$PRAGMA IGNORE_TKR buf !DIR\$ IGNORE_TKR buf
43 44 45	<pre>!IBM* IGNORE_TKR buf REAL, DIMENSION(*) :: buf ! declarations of the other arguments END SUBROUTINE</pre>
46 47	END INTERFACE
48	(End of advice to implementors)

(End of advice to implementors.)

19.1.7 Requirements on Fortran Compilers

 $\mathsf{MPI-3.0}$ (and later) compliant Fortran bindings are not only a property of the MPI library itself, but rather a property of an MPI library together with the Fortran compiler suite for which it is compiled.

Advice to users. Users must take appropriate steps to ensure that proper options are specified to compilers. MPI libraries must document these options. Some MPI libraries are shipped together with special compilation scripts (e.g., mpif90, mpicc) that set these options automatically. (End of advice to users.)

An MPI library together with the Fortran compiler suite is only compliant with MPI-3.0 (and later), as referred by MPI_GET_VERSION, if all the solutions described in Sections 19.1.11 through 19.1.19 work correctly. Based on this rule, major requirements for all three Fortran support methods (i.e., the mpi_f08 and mpi modules, and mpif.h) are:

- The language features assumed-type and assumed-rank from Fortran 2008 TS 29113 [46] are available. This is required only for mpi_f08. As long as this requirement is not supported by the compiler, it is valid to build an MPI library that implements the mpi_f08 module with MPI_SUBARRAYS_SUPPORTED set to .FALSE..
- "Simply contiguous" arrays and scalars must be passed to choice buffer dummy arguments of nonblocking routines with call by reference. This is needed only if one of the support methods does not use the ASYNCHRONOUS attribute. See Section 19.1.12 for more details.
- SEQUENCE and BIND(C) derived types are valid as actual arguments passed to choice buffer dummy arguments, and, in the case of MPI_SUBARRAYS_SUPPORTED== .FALSE., they are passed with call by reference, and passed by descriptor in the case of .TRUE..
- All actual arguments that are allowed for a dummy argument in an implicitly defined and separately compiled Fortran routine with the given compiler (e.g., CHARACTER(LEN=*) strings and array of strings) must also be valid for choice buffer dummy arguments with all Fortran support methods.
- The array dummy argument of the ISO_C_BINDING intrinsic module procedure C_F_POINTER is not restricted to Fortran types for which a corresponding type in C exists.
- The Fortran compiler shall not provide TYPE(*) unless the ASYNCHRONOUS attribute protects MPI communication as described in TS 29113. Specifically, the TS 29113 must be implemented as a whole.

The following rules are required at least as long as the compiler does not provide the extension of the ASYNCHRONOUS attribute as part of TS 29113 and there still exists a Fortran support method with MPI_ASYNC_PROTECTS_NONBLOCKING==.FALSE.. Observation of these rules by the MPI application developer is especially recomended for backward compatibility of existing applications that use the mpi module or the mpif.h include file. The rules are as follows: $\overline{7}$

 24

- Separately compiled empty Fortran routines with implicit interfaces and separately compiled empty C routines with BIND(C) Fortran interfaces (e.g., MPI_F_SYNC_REG on page 770 and Section 19.1.8, and DD on page 771) solve the problems described in Section 19.1.17.
 - The problems with temporary data movement (described in detail in Section 19.1.18) are solved as long as the application uses different sets of variables for the nonblocking communication (or nonblocking or split collective I/O) and the computation when overlapping communication and computation.
 - Problems caused by automatic and permanent data movement (e.g., within a garbage collection, see Section 19.1.19) are resolved **without** any further requirements on the application program, neither on the usage of the buffers, nor on the declaration of application routines that are involved in invoking MPI procedures.

All of these rules are valid for the mpi_f08 and mpi modules and independently of whether mpif.h uses explicit interfaces.

Advice to implementors. Some of these rules are already part of the Fortran 2003 standard, some of these requirements require the Fortran TS 29113 [46], and some of these requirements for MPI-3.0 are beyond the scope of TS 29113. (End of advice to implementors.)

 24

19.1.8 Additional Support for Fortran Register-Memory-Synchronization

As described in Section 19.1.17, a dummy call may be necessary to tell the compiler that registers are to be flushed for a given buffer or that accesses to a buffer may not be moved across a given point in the execution sequence. Only a Fortran binding exists for this call.

- ²⁹ MPI_F_SYNC_REG(buf)
 - INOUT buf

initial address of buffer (choice)

Fortran 2008 binding

MPI_F_sync_reg(buf)

TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf

³⁶ Fortran binding
 ³⁷ MPI_F_SYNC_REG(BUF)
 ³⁸ <type> BUF(*)

This routine has no executable statements. It must be compiled in the MPI library in such a manner that a Fortran compiler cannot detect in the module that the routine has an empty body. It is used only to force the compiler to flush a cached register value of a variable or buffer back to memory (when necessary), or to invalidate the register value.

Rationale. This function is not available in other languages because it would not be useful. This routine has no ierror return argument because there is no operation that can fail. (*End of rationale.*)

 $\mathbf{2}$

Advice to implementors. This routine can be bound to a C routine to minimize the risk that the Fortran compiler can learn that this routine is empty (and that the call to this routine can be removed as part of an optimization). However, it is explicitly allowed to implement this routine within the mpi_f08 module according to the definition for the mpi module or mpif.h to circumvent the overhead of building the internal dope vector to handle the assumed-type, assumed-rank argument. (End of advice to implementors.)

Rationale. This routine is not defined with TYPE(*), DIMENSION(*), i.e., assumed size instead of assumed rank, because this would restrict the usability to "simply contiguous" arrays and would require overloading with another interface for scalar arguments. (End of rationale.)

If only a part of an array (e.g., defined by a subscript triplet) is Advice to users. used in a nonblocking routine, it is recommended to pass the whole array to MPI_F_SYNC_REG anyway to minimize the overhead of this no-operation call. Note that this routine need not be called if MPI_ASYNC_PROTECTS_NONBLOCKING is .TRUE. and the application fully uses the facilities of ASYNCHRONOUS arrays. (End of advice to users.)

Additional Support for Fortran Numeric Intrinsic Types 19.1.9

MPI provides a small number of named datatypes that correspond to named intrinsic types supported by C and Fortran. These include MPI_INTEGER, MPI_REAL, MPI_INT, MPI_DOUBLE, etc., as well as the optional types MPI_REAL4, MPI_REAL8, etc. There is a one-to-one correspondence between language declarations and MPI types.

Fortran (starting with Fortran 90) provides so-called KIND-parameterized types. These types are declared using an intrinsic type (one of INTEGER, REAL, COMPLEX, LOGICAL, and 27CHARACTER) with an optional integer KIND parameter that selects from among one or more 2829variants. The specific meaning of different KIND values themselves are implementation 30 dependent and not specified by the language. Fortran provides the KIND selection functions 31selected_real_kind for REAL and COMPLEX types, and selected_int_kind for INTEGER types that allow users to declare variables with a minimum precision or number of digits. 33 These functions provide a portable way to declare KIND-parameterized REAL, COMPLEX, and 34 INTEGER variables in Fortran. This scheme is backward compatible with Fortran 77. REAL and INTEGER Fortran variables have a default KIND if none is specified. Fortran DOUBLE 3536 PRECISION variables are of intrinsic type REAL with a non-default KIND. The following two 37 declarations are equivalent:

double precision x real(KIND(0.0d0)) x

41 MPI provides two orthogonal methods for handling communication buffers of numeric 42intrinsic types. The first method (see the following section) can be used when variables have been declared in a portable way — using default KIND or using KIND parameters obtained 4344with the selected_int_kind or selected_real_kind functions. With this method, MPI automatically selects the correct data size (e.g., 4 or 8 bytes) and provides representation 4546conversion in heterogeneous environments. The second method (see "Support for size-47specific MPI Datatypes" on page 754) gives the user complete control over communication 48 by exposing machine representations.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

 $\overline{7}$

8

9

10

11

1213

14

15

16

17

18

1920

21

22

23 24

25

26

32

38

39

Parameterized Datatypes with Specified Precision and Exponent Range

 $^2_{_3}$ MPI provides named datatypes corresponding to standard Fortran 77 numeric types:

⁴ MPI_INTEGER, MPI_COMPLEX, MPI_REAL, MPI_DOUBLE_PRECISION and

⁵ MPI_DOUBLE_COMPLEX. MPI automatically selects the correct data size and provides rep-⁶ resentation conversion in heterogeneous environments. The mechanism described in this ⁷ section extends this model to support portable parameterized numeric types.

The model for supporting portable parameterized types is as follows. Real variables 8 are declared (perhaps indirectly) using selected_real_kind(p, r) to determine the KIND 9 parameter, where \mathbf{p} is decimal digits of precision and \mathbf{r} is an exponent range. Implicitly 10 MPI maintains a two-dimensional array of predefined MPI datatypes D(p, r). D(p, r) is 11 defined for each value of (p, r) supported by the compiler, including pairs for which one 12value is unspecified. Attempting to access an element of the array with an index (p, r) not 13 supported by the compiler is erroneous. MPI implicitly maintains a similar array of COMPLEX 14 datatypes. For integers, there is a similar implicit array related to selected_int_kind and 15 indexed by the requested number of digits r. Note that the predefined datatypes contained 16 in these implicit arrays are not the same as the named MPI datatypes MPI_REAL, etc., but 17a new set. 18

Advice to implementors. The above description is for explanatory purposes only. It is not expected that implementations will have such internal arrays. (End of advice to implementors.)

Advice to users. selected_real_kind() maps a large number of (p,r) pairs to a much smaller number of KIND parameters supported by the compiler. KIND parameters are not specified by the language and are not portable. From the language point of view intrinsic types of the same base type and KIND parameter are of the same type. In order to allow interoperability in a heterogeneous environment, MPI is more stringent. The corresponding MPI datatypes match if and only if they have the same (p,r) value (REAL and COMPLEX) or r value (INTEGER). Thus MPI has many more datatypes than there are fundamental language types. (End of advice to users.)

32 33 34

45

19

20

21

22 23

24

25

26

27

28

29

30

31

MPI_TYPE_CREATE_F90_REAL(p, r, newtype)

IN	р	precision, in decimal digits (integer)
IN	r	decimal exponent range (integer)
OUT	newtype	the requested MPI data type (handle)

⁴⁰ C binding

```
<sup>41</sup> int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
```

```
42
43 Fortran 2008 binding
```

```
MPI_Type_create_f90_real(p, r, newtype, ierror)
```

```
INTEGER, INTENT(IN) :: p, r
```

```
46 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
```

```
47 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

⁴⁸ Fortran binding

MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR) INTEGER P, R, NEWTYPE, IERROR

This function returns a predefined MPI datatype that matches a REAL variable of KIND selected_real_kind(p, r). In the model described above it returns a handle for the element D(p, r). Either p or r may be omitted from calls to selected_real_kind(p, r) (but not both). Analogously, either p or r may be set to MPI_UNDEFINED. In communication, an MPI datatype A returned by MPI_TYPE_CREATE_F90_REAL matches a datatype B if and only if B was returned by MPI_TYPE_CREATE_F90_REAL called with the same values for p and r or B is a duplicate of such a datatype. Restrictions on using the returned datatype with the "external32" data representation are given on page 753.

It is erroneous to supply values for \boldsymbol{p} and \boldsymbol{r} not supported by the compiler.

MPI_TYPE_CREATE_F90_COMPLEX(p, r, newtype)

IN	р	precision, in decimal digits (integer)
IN	r	decimal exponent range (integer)
OUT	newtype	the requested MPI datatype (handle)

C binding

int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)

Fortran 2008 binding

<pre>MPI_Type_create_f90_complex(p, r, newtype, ierror)</pre>	
INTEGER, INTENT(IN) :: p, r	
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	

Fortran binding

```
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
INTEGER P, R, NEWTYPE, IERROR
```

This function returns a predefined MPI datatype that matches a COMPLEX variable of KIND selected_real_kind(p, r). Either p or r may be omitted from calls to selected_real_kind(p, r) (but not both). Analogously, either p or r may be set to MPI_UNDEFINED. Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions on using the returned datatype with the "external32" data representation are given on page 753.

It is erroneous to supply values for \boldsymbol{p} and \boldsymbol{r} not supported by the compiler.

MPI_TYPE_CREATE_F90_INTEGER(r, newtype) IN r decimal exponent range, i.e., number of decimal digits (integer) OUT newtype the requested MPI datatype (handle)

C binding

Unofficial Draft for Comment Only

 $\mathbf{2}$

```
1
     int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
2
     Fortran 2008 binding
3
     MPI_Type_create_f90_integer(r, newtype, ierror)
4
          INTEGER, INTENT(IN) :: r
5
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
6
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     Fortran binding
9
     MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
10
          INTEGER R, NEWTYPE, IERROR
11
         This function returns a predefined MPI datatype that matches a INTEGER variable of
12
     KIND selected_int_kind(r). Matching rules for datatypes created by this function are
13
     analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL.
14
     Restrictions on using the returned datatype with the "external 32" data representation are
15
     given on page 753.
16
         It is erroneous to supply a value for r that is not supported by the compiler.
17
         Example:
18
19
     integer
                     longtype, quadtype
20
     integer, parameter :: long = selected_int_kind(15)
21
     integer(long) ii(10)
22
     real(selected_real_kind(30)) x(10)
23
     call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
24
     call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
25
     . . .
26
27
     call MPI_SEND(ii, 10, longtype, ...)
28
     call MPI_SEND(x, 10, quadtype, ...)
29
30
                              The datatypes returned by the above functions are predefined
           Advice to users.
31
           datatypes. They cannot be freed; they do not need to be committed; they can be
32
           used with predefined reduction operations. There are two situations in which they
33
           behave differently syntactically, but not semantically, from the MPI named predefined
34
           datatypes.
35
             1. MPI_TYPE_GET_ENVELOPE returns special combiners that allow a program to
36
               retrieve the values of p and r.
37
38
            2. Because the datatypes are not named, they cannot be used as compile-time
39
               initializers or otherwise accessed before a call to one of the
40
               MPI_TYPE_CREATE_F90_XXX routines.
41
           If a variable was declared specifying a non-default KIND value that was not obtained
42
           with selected_real_kind() or selected_int_kind(), the only way to obtain a
43
           matching MPI datatype is to use the size-based mechanism described in the next
44
           section.
45
46
           (End of advice to users.)
47
48
```

Advice to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_XXX with the same combination of (XXX,p,r). The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, a high quality MPI implementation should return the same datatype handle for the same (REAL/COMPLEX/ INTEGER,p,r) combination. Checking for the combination (p,r) in the preceding call to MPI_TYPE_CREATE_F90_XXX and using a hash table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,p,r). (*End of advice to implementors.*)

Rationale. The MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER interface needs as input the original range and precision values to be able to define useful and compiler-independent external (Section 14.5.2) or user-defined (Section 14.5.3) data representations, and in order to be able to perform automatic and efficient data conversions in a heterogeneous environment. (*End of rationale.*)

We now specify how the datatypes described in this section behave when used with the "external32" external data representation described in Section 14.5.2.

The external32 representation specifies data formats for integer and floating point values. Integer values are represented in two's complement big-endian format. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single," "Double," and "Double Extended" formats, requiring 4, 8, and 16 bytes of storage, respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +10383, 112 fraction bits, and an encoding analogous to the "Double" format.

The external32 representations of the datatypes returned by MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER are given by the following rules. For MPI_TYPE_CREATE_F90_REAL:

if $(p > 33)$ or $(r > 4931)$ then external32 representation	29
is undefined	30
else if $(p > 15)$ or $(r > 307)$ then external32_size = 16	31
else if $(p > 6)$ or $(r > 37)$ then external32_size = 8	32
else external32_size = 4	33
	34
For MPI_TYPE_CREATE_F90_COMPLEX: twice the size as for	35
MPI_TYPE_CREATE_F90_REAL.	36
For MPI_TYPE_CREATE_F90_INTEGER:	37
	38
if $(r > 38)$ then external32 representation is undefined	39
else if $(r > 18)$ then external32_size = 16	40
else if (r > 9) then external32_size = 8	41
else if $(r > 4)$ then external32_size = 4	42
else if $(r > 2)$ then external32_size = 2	43
else external32_size = 1	43
	44

If the external 32 representation of a datatype is undefined, the result of using the datatype ⁴⁵ directly or indirectly (i.e., as part of another datatype or through a duplicated datatype) ⁴⁶ in operations that require the external 32 representation is undefined. These operations include MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL, and many MPI_FILE functions, ⁴⁸

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8 9

10

11

12

13

14

15 16

17

18

19

20

21

22

23

 24

25

26

27

1 when the "external32" data representation is used. The ranges for which the external32 $\mathbf{2}$ representation is undefined are reserved for future standardization.

3 4

5

Support for Size-specific MPI Datatypes

MPI provides named datatypes corresponding to optional Fortran 77 numeric types that 6 contain explicit byte lengths — MPI_REAL4, MPI_INTEGER8, etc. This section describes a 7 mechanism that generalizes this model to support all Fortran numeric intrinsic types. 8

We assume that for each **typeclass** (integer, real, complex) and each word size there is 9 a unique machine representation. For every pair (typeclass, \mathbf{n}) supported by a compiler, 10 MPI must provide a named size-specific datatype. The name of this datatype is of the form 11 MPI_<TYPE>n in C and Fortran where <TYPE> is one of REAL, INTEGER and COMPLEX, and 12**n** is the length in bytes of the machine representation. This datatype locally matches all 13 variables of type (typeclass, n) in Fortran. The list of names for such types includes: 14

- 15MPI_REAL4
- 16MPI_REAL8
- 17MPI_REAL16
- 18 MPI_COMPLEX8 19
- MPI_COMPLEX16 20
- MPI_COMPLEX32 21
- MPI_INTEGER1 22
- MPI_INTEGER2 23
- MPI_INTEGER4 24
- MPI_INTEGER8 25
- MPI_INTEGER16 26

27One datatype is required for each representation supported by the Fortran compiler.

Rationale. Particularly for the longer floating-point types, C and Fortran may use different representations. For example, a Fortran compiler may define a 16-byte REAL type with 33 decimal digits of precision while a C compiler may define a 16-byte long double type that implements an 80-bit (10 byte) extended precision floating point value. Both of these types are 16 bytes long, but they are not interoperable. Thus, these types are defined by Fortran, even though C may define types of the same length. (End of rationale.)

To be backward compatible with the interpretation of these types in MPI-1, we assume that the nonstandard declarations REAL*n, INTEGER*n, always create a variable whose rep-38 resentation is of size \mathbf{n} . These datatypes may also be used for variables declared with 39 KIND=INT8/16/32/64 or KIND=REAL32/64/128, which are defined in the ISO_FORTRAN_ENV 40 intrinsic module. Note that the MPI datatypes and the REAL*n, INTEGER*n declarations count bytes whereas the Fortran KIND values count bits. All these datatypes are predefined. 42

43

41

28

29

30

31

32

33 34

3536

- 4445
- 46
- 47
- 48

MPI_TYPE	_MATCH_SIZE(typeclass, size	, datatype)	1
IN	typeclass	generic type specifier (integer)	2
IN	size	size, in bytes, of representation (integer)	$\frac{3}{4}$
Ουτ	datatype	datatype with correct type, size (handle)	5
001	datatype	datatype with correct type, size (nandle)	6
C binding	r		7
-	-	ass, int size, MPI_Datatype *datatype)	8
Fortnon 9	008 hinding		9
	008 binding match_size(typeclass, siz	e datatupe jerror)	10 11
	ER, INTENT(IN) :: typecla		11
	MPI_Datatype), INTENT(OUT		13
	ER, OPTIONAL, INTENT(OUT)		14
Fortran b	inding		15
	MATCH_SIZE(TYPECLASS, SIZ	E, DATATYPE, IERROR)	16
	ER TYPECLASS, SIZE, DATAT		17
			18
• •		_REAL, MPI_TYPECLASS_INTEGER and ng to the desired typeclass . The function returns	19
	atype matching a local variab		20 21
		andle) to one of the predefined named datatypes,	21
		freed. MPI_TYPE_MATCH_SIZE can be used to	23
_		a Fortran numeric intrinsic type by first calling	24
storage_size	e() in order to compute the va	ariable size in bits, dividing it by eight, and then	25
		d a suitable datatype. In C, one can use the C	26
		in bytes) instead of storage_size() (which returns	27
		of default kind the variable's size can be computed	28
	supported by the compiler.	the typeclass is known. It is erroneous to specify	29 30
a size not a	supported by the complier.		31
Ratic	<i>nale.</i> This is a convenience f	function. Without it, it can be tedious to find the	32
corre	ct named type. See note to im	plementors below. (End of rationale.)	33
			34
Advie	ce to implementors. This fund	ction could be implemented as a series of tests.	35
int 1	MDI Turne metch size(int t	uncologa int size MDI Detetune trtune)	36
1110 I {	WP1_Type_match_size(int t	ypeclass, int size, MPI_Datatype *rtype)	37
-	<pre>itch(typeclass) {</pre>		38
	case MPI_TYPECLASS_REAL	: switch(size) {	39 40
	<pre>case 4: *rtype = MPI_</pre>	REAL4; return MPI_SUCCESS;	41
	case 8: *rtype = MPI_	REAL8; return MPI_SUCCESS;	42
	<pre>default: error();</pre>		43
			44
	case MPI_TYPECLASS_INTE		45
	• 1	_INTEGER4; return MPI_SUCCESS; _INTEGER8; return MPI_SUCCESS;	46
	<pre>default: error();</pre>		47
			48

```
1
                  }
2
                 ... etc. ...
3
              }
4
5
              return MPI_SUCCESS;
6
           }
7
8
           (End of advice to implementors.)
9
10
     Communication With Size-specific Types
11
     The usual type matching rules apply to size-specific datatypes: a value sent with datatype
12
     MPI_{TYPE>n} can be received with this same datatype on another process. Most modern
13
     computers use 2's complement for integers and IEEE format for floating point. Thus, com-
14
     munication using these size-specific datatypes will not entail loss of precision or truncation
15
     errors.
16
17
           Advice to users. Care is required when communicating in a heterogeneous environ-
18
           ment. Consider the following code:
19
20
           real(selected_real_kind(5)) x(100)
21
           size = storage_size(x) / 8 •
22
           call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
23
           if (myrank .eq. 0) then
24
                ... initialize x ...
25
                call MPI_SEND(x, xtype, 100, 1,
                                                     ...)
26
           else if (myrank .eq. 1) then
27
                call MPI_RECV(x, xtype, 100, 0,
28
           endif
29
30
           This may not work in a heterogeneous environment if the value of size is not the
31
           same on process 1 and process 0. There should be no problem in a homogeneous
32
           environment. To communicate in a heterogeneous environment, there are at least four
33
           options. The first is to declare variables of default type and use the MPI datatypes
34
           for these types, e.g., declare a variable of type REAL and use MPI_REAL. The second
35
           is to use selected_real_kind or selected_int_kind and with the functions of the
36
           previous section. The third is to declare a variable that is known to be the same
37
           size on all architectures (e.g., selected_real_kind(12) on almost all compilers will
38
           result in an 8-byte representation). The fourth is to carefully check representation
39
           size before communication. This may require explicit conversion to a variable of size
40
           that can be communicated and handshaking between sender and receiver to agree on
41
           a size.
42
           Note finally that using the "external32" representation for I/O requires explicit at-
43
           tention to the representation sizes. Consider the following code:
44
45
46
           real(selected_real_kind(5)) x(100)
47
           size = storage_size(x) / 8
48
           call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
```

```
if (myrank .eq. 0) then
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',
                                                            &
                      MPI_MODE_CREATE+MPI_MODE_WRONLY,
                                                            &
                      MPI_INFO_NULL, fh, ierror)
   call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32',&
                          MPI_INFO_NULL, ierror)
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
   call MPI_FILE_CLOSE(fh, ierror)
endif
call MPI_BARRIER(MPI_COMM_WORLD, ierror)
if (myrank .eq. 1) then
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY,
                                                                X.
                 MPI_INFO_NULL, fh, ierror)
   call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32',&
                          MPI_INFO_NULL, ierror)
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
   call MPI_FILE_CLOSE(fh, ierror)
endif
```

If processes 0 and 1 are on different machines, this code may not work as expected if the size is different on the two machines. (*End of advice to users.*)

19.1.10 Problems With Fortran Bindings for MPI

This section discusses a number of problems that may arise when using MPI in a Fortran program. It is intended as advice to users, and clarifies how MPI interacts with Fortran. It is intended to clarify, not add to, this standard.

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these may cause few problems for Fortran 77 programs, they become more significant for Fortran 90 programs, so that users must exercise care when using new Fortran 90 features. With Fortran 2008 and the new semantics defined in TS 29113, most violations are resolved, and this is hinted at in an addendum to each item. The violations were originally adopted and have been retained because they are important for the usability of MPI. The rest of this section describes the potential problems in detail.

The following MPI features are inconsistent with Fortran 90 and Fortran 77.

- 1. An MPI subroutine with a choice argument may be called with different argument types. When using the mpi_f08 module together with a compiler that supports Fortran 2008 + TS 29113, this problem is resolved.
- 2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument. This is only solved for choice buffers through the use of DIMENSION(...).
- 3. Nonblocking and split-collective MPI routines assume that actual arguments are passed by address or descriptor and that arguments and the associated data are not copied

Unofficial Draft for Comment Only

2

6

9 10

11

12 13 14

15

16

17

18

19

20 21

22 23

 24

25 26

27 28

29

30

31

32

33

34

35

36

37

38

39

40

41

42 43

44

45

12	on entrance to or exit from the subroutine. This problem is solved with the use of the ASYNCHRONOUS attribute.
3 4 5 6 7	4. An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program that is executing outside of MPI calls. This problem is resolved by relying on the extended semantics of the ASYNCHRONOUS attribute as specified in TS 29113.
8 9 10 11 12 13	5. Several named "constants," such as MPI_BOTTOM, MPI_IN_PLACE, MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE, MPI_UNWEIGHTED, MPI_WEIGHTS_EMPTY, MPI_ARGV_NULL, and MPI_ARGVS_NULL are not ordinary Fortran constants and require a special implementation. See Sec- tion 2.5.4 for more information.
14 15 16 17 18 19 20 21	 6. The memory allocation routine MPI_ALLOC_MEM cannot be used from Fortran 77/90/95 without a language extension (for example, Cray pointers) that allows the allocated memory to be associated with a Fortran variable. Therefore, address sized integers were used in MPI-2.0 - MPI-2.2. In Fortran 2003, TYPE(C_PTR) entities were added, which allow a standard-conforming implementation of the semantics of MPI_ALLOC_MEM. In MPI-3.0 and later, MPI_ALLOC_MEM has an additional, overloaded interface to support this language feature. The use of Cray pointers is deprecated. The mpi_f08 module only supports TYPE(C_PTR) pointers.
22 23	Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.
24 25	• MPI identifiers exceed 6 characters.
26	• MPI identifiers may contain underscores after the first character.
27 28 29	• MPI requires an include file, mpif.h. On systems that do not support include files, the implementation should specify the values of named constants.
30 31 32 33 34	• Many routines in MPI have KIND-parameterized integers (e.g., MPI_ADDRESS_KIND and MPI_OFFSET_KIND) that hold address information. On systems that do not support Fortran 90-style parameterized types, INTEGER*8 or INTEGER should be used instead.
34 35	MPI-1 contained several routines that take address-sized information as input or return
36	address-sized information as output. In C such arguments were of type
37 38	MPI_Aint and in Fortran of type INTEGER. On machines where integers are smaller than addresses, these routines can less information. In MPI 2 the use of these functions has
39	addresses, these routines can lose information. In MPI-2 the use of these functions has been deprecated and they have been replaced by routines taking INTEGER arguments of
	KIND=MPI_ADDRESS_KIND. A number of MPI-2 functions also take INTEGER arguments of
40	KIND-FFI_ADDRESS_KIND. A HUMBER OF WFF-2 TUBCTORS also take INTEGER alguments of
40 41	non-default KIND. See Section 2.6 and Section 5.1.1 for more information.
	non-default KIND. See Section 2.6 and Section 5.1.1 for more information. Sections 19.1.11 through 19.1.19 describe several problems in detail which concern
41 42 43	non-default KIND. See Section 2.6 and Section 5.1.1 for more information. Sections 19.1.11 through 19.1.19 describe several problems in detail which concern the interaction of MPI and Fortran as well as their solutions. Some of these solutions
41 42	 non-default KIND. See Section 2.6 and Section 5.1.1 for more information. Sections 19.1.11 through 19.1.19 describe several problems in detail which concern the interaction of MPI and Fortran as well as their solutions. Some of these solutions require special capabilities from the compilers. Major requirements are summarized in
41 42 43 44	non-default KIND. See Section 2.6 and Section 5.1.1 for more information. Sections 19.1.11 through 19.1.19 describe several problems in detail which concern the interaction of MPI and Fortran as well as their solutions. Some of these solutions
41 42 43 44 45	 non-default KIND. See Section 2.6 and Section 5.1.1 for more information. Sections 19.1.11 through 19.1.19 describe several problems in detail which concern the interaction of MPI and Fortran as well as their solutions. Some of these solutions require special capabilities from the compilers. Major requirements are summarized in

19.1.11 Problems Due to Strong Typing

All MPI functions with choice arguments associate actual arguments of different Fortran datatypes with the same dummy argument. This is not allowed by Fortran 77, and in Fortran 90, it is technically only allowed if the function is overloaded with a different function for each type (see also Section 19.1.6). In C, the use of void* formal arguments avoids these problems. Similar to C, with Fortran 2008 + TS 29113 (and later) together with the mpi_f08 module, the problem is avoided by declaring choice arguments with TYPE(*), DIMENSION(..), i.e., as assumed-type and assumed-rank dummy arguments.

Using INCLUDE 'mpif.h', the following code fragment is technically invalid and may generate a compile-time error.

```
integer i(5)
real x(5)
...
call mpi_send(x, 5, MPI_REAL, ...)
call mpi_send(i, 5, MPI_INTEGER, ...)
```

In practice, it is rare for compilers to do more than issue a warning. When using either the mpi_f08 or mpi module, the problem is usually resolved through the assumed-type and assumed-rank declarations of the dummy arguments, or with a compiler-dependent mechanism that overrides type checking for choice arguments.

It is also technically invalid in Fortran to pass a scalar actual argument to an array dummy argument that is not a choice buffer argument. Thus, when using the mpi_f08 or mpi module, the following code fragment usually generates an error since the dims and periods arguments to MPI_CART_CREATE are declared as assumed size arrays INTEGER :: DIMS(*) and LOGICAL :: PERIODS(*).

USE mpi_f08 ! or USE mpi					
INTEGER size					
CALL MPI_Cart_create(comm_old,	1, si	ze, .TRUE.,	.TRUE.,	comm_cart,	ierror)

Although this is a non-conforming MPI call, compiler warnings are not expected (but may occur) when using INCLUDE 'mpif.h' and this include file does not use Fortran explicit interfaces.

19.1.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets

Arrays with subscript triplets describe Fortran subarrays with or without strides, e.g.,

```
REAL a(100,100,100)
CALL MPI_Send(a(11:17, 12:99:3, 1:100), 7*30*100, MPI_REAL, ...)
```

The handling of subscript triplets depends on the value of the constant MPI_SUBARRAYS_SUPPORTED:

• If MPI_SUBARRAYS_SUPPORTED equals .TRUE.:

Choice buffer arguments are declared as TYPE(*), DIMENSION(..). For example, consider the following code fragment:

Unofficial Draft for Comment Only

 24

1	REAL s(100), r(100)
2	CALL MPI_Isend(s(1:100:5), 3, MPI_REAL,, rq, ierror)
3	CALL MPI_Wait(rq, status, ierror)
4	CALL MPI_Irecv(r(1:100:5), 3, MPI_REAL,, rq, ierror)
5	CALL MPI_Wait(rq, status, ierror)
6	
7	In this case, the individual elements $s(1)$, $s(6)$, and $s(11)$ are sent between the start
8	
9	of MPI_ISEND and the end of MPI_WAIT even though the compiled code will not copy
10	s(1:100:5) to a real contiguous temporary scratch buffer. Instead, the compiled code
11	will pass a descriptor to MPI_ISEND that allows MPI to operate directly on s(1), s(6),
12	$s(11), \ldots, s(96)$. The called MPI_ISEND routine will take only the first three of these
13	elements due to the type signature "3, MPI_REAL".
14	All nonblocking MPI functions (e.g., MPI_ISEND, MPI_PUT,
14	MPI_FILE_WRITE_ALL_BEGIN) behave as if the user-specified elements of choice buf-
	fers are copied to a contiguous scratch buffer in the MPI runtime environment. All
16	datatype descriptions (in the example above, "3, MPI_REAL") read and store data
17	from and to this virtual contiguous scratch buffer. Displacements in MPI derived
18	datatypes are relative to the beginning of this virtual contiguous scratch buffer. Upon
19	completion of a nonblocking receive operation (e.g., when MPI_WAIT on a correspond-
20	ing MPI_Request returns), it is as if the received data has been copied from the virtual
21	contiguous scratch buffer back to the non-contiguous application buffer. In the ex-
22	
23	ample above, $r(1)$, $r(6)$, and $r(11)$ are guaranteed to be defined with the received data when MPL WALT returns
24	data when MPI_WAIT returns.
25	Note that the above definition does not supercede restrictions about buffers used with
26	nonblocking operations (e.g., those specified in Section $3.7.2$).
27	
28	Advice to implementors. The Fortran descriptor for TYPE(*), DIMENSION()
29	arguments contains enough information that, if desired, the MPI library can make
30	a real contiguous copy of non-contiguous user buffers when the nonblocking op-
31	eration is started, and release this buffer not before the nonblocking communi-
32	cation has completed (e.g., the MPI_WAIT routine). Efficient implementations
33	may avoid such additional memory-to-memory data copying. (End of advice to
34	implementors.)
35	
36	Rationale. If MPI_SUBARRAYS_SUPPORTED equals .TRUE., non-contiguous
37	buffers are handled inside the MPI library instead of by the compiler through
38	argument association conventions. Therefore, the scope of MPI library scratch
39	buffers can be from the beginning of a nonblocking operation until the completion
40	of the operation although beginning and completion are implemented in different
41	routines. (End of rationale.)
42	
42	• If MPI_SUBARRAYS_SUPPORTED equals .FALSE.:
	In this case, the use of Fortran arrays with subscript triplets as actual choice buffer
44	
45	arguments in any nonblocking MPI operation (which also includes persistent request,
46	and split collectives) may cause undefined behavior. They may, however, be used in
47	blocking MPI operations.

Implicit in MPI is the idea of a contiguous chunk of memory accessible through a linear address space. MPI copies data to and from this memory. An MPI program specifies the location of data by providing memory addresses and offsets. In the C language, sequence association rules plus pointers provide all the necessary low-level structure.

In Fortran, array data is not necessarily stored contiguously. For example, the array section A(1:N:2) involves only the elements of A with indices 1, 3, 5, The same is true for a pointer array whose target is such a section. Most compilers ensure that an array that is a dummy argument is held in contiguous memory if it is declared with an explicit shape (e.g., B(N)) or is of assumed size (e.g., B(*)). If necessary, they do this by making a copy of the array into contiguous memory.¹

Because MPI dummy buffer arguments are assumed-size arrays if MPI_SUBARRAYS_SUPPORTED equals .FALSE., this leads to a serious problem for a nonblocking call: the compiler copies the temporary array back on return but MPI continues to copy data to the memory that held it. For example, consider the following code fragment:

real a(100)
call MPI_IRECV(a(1:100:2), MPI_REAL, 50, ...)

Since the first dummy argument to MPI_IRECV is an assumed-size array (<type> buf(*)), the array section a(1:100:2) is copied to a temporary before being passed to MPI_IRECV, so that it is contiguous in memory. MPI_IRECV returns immediately, and data is copied from the temporary back into the array a. Sometime later, MPI may write to the address of the deallocated temporary. Copying is also a problem for MPI_ISEND since the temporary array may be deallocated before the data has all been sent from it.

Most Fortran 90 compilers do not make a copy if the actual argument is the whole of an explicit-shape or assumed-size array or is a "simply contiguous" section such as A(1:N) of such an array. ("Simply contiguous" is defined in the next paragraph.) Also, many compilers treat allocatable arrays the same as they treat explicit-shape arrays in this regard (though we know of one that does not). However, the same is not true for assumed-shape and pointer arrays; since they may be discontiguous, copying is often done. It is this copying that causes problems for MPI as described in the previous paragraph.

According to the Fortran 2008 Standard, Section 6.5.4, a "simply contiguous" array section is

That is, there are zero or more dimensions that are selected in full, then one dimension selected without a stride, then zero or more dimensions that are selected with a simple subscript. The compiler can detect from analyzing the source code that the array is contiguous. Examples are $\mathbf{2}$

 $\overline{7}$

¹Technically, the Fortran standard is worded to allow non-contiguous storage of any array data, unless the dummy argument has the CONTIGUOUS attribute.

1	A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:,:,1:N)
2	
3 4	Because of Fortran's column-major ordering, where the first index varies fastest, a "simply contiguous" section of a contiguous array will also be contiguous.
5	The same problem can occur with a scalar argument. A compiler may make a copy of
6	scalar dummy arguments within a called procedure when passed as an actual argument
7	to a choice buffer routine. That this can cause a problem is illustrated by the example
8	
9	
10	real :: a
11	call user1(a,rq)
12	call MPI_WAIT(rq,status,ierr)
13	write (*,*) a
14	
15	subroutine user1(buf,request)
16	call MPI_IRECV(buf,,request,)
17	end
18	
19	If a is copied, MPI_IRECV will alter the copy when it completes the communication
20	and will not alter a itself.
21	Note that copying will almost certainly occur for an argument that is a non-trivial
22	expression (one with at least one operator or function call), a section that does not
23 24	select a contiguous part of its parent (e.g., A(1:n:2)), a pointer whose target is such
24	a section, or an assumed-shape array that is (directly or indirectly) associated with
26	such a section.
27	If a compiler option exists that inhibits copying of arguments, in either the calling or
28	called procedure, this must be employed.
29	If a compiler makes copies in the calling procedure of arguments that are explicit-
30	shape or assumed-size arrays, "simply contiguous" array sections of such arrays, or
31	scalars, and if no compiler option exists to inhibit such copying, then the compiler
32	cannot be used for applications that use MPI_GET_ADDRESS, or any nonblocking
33	MPI routine. If a compiler copies scalar arguments in the called procedure and there
34	is no compiler option to inhibit this, then this compiler cannot be used for applications
35	that use memory references across subroutine calls as in the example above.
36	
37 38	19.1.13 Problems Due to Data Copying and Sequence Association with Vector Subscripts
39	Fortran arrays with vector subscripts describe subarrays containing a possibly irregular
40	set of elements
41	
42	REAL a(100)
43	CALL MPI_Send(A((/7,9,23,81,82/)), 5, MPI_REAL,)
44	
45	Fortran arrays with a vector subscript must not be used as actual choice buffer argu-
46	ments in any nonblocking or split collective MPI operations. They may, however, be used
47	in blocking MPI operations.
48	

19.1.14 Special Constants

MPI requires a number of special "constants" that cannot be implemented as normal Fortran constants, e.g., MPI_BOTTOM. The complete list can be found in Section 2.5.4. In C, these are implemented as constant pointers, usually as NULL and are used where the function prototype calls for a pointer to a variable, not the variable itself.

In Fortran, using special values for the constants (e.g., by defining them through **parameter** statements) is not possible because an implementation cannot distinguish these values from valid data. Typically these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared COMMON block), relying on the fact that the target compiler passes data by address. Inside the subroutine, the address of the actual choice buffer argument can be compared with the address of such a predefined static variable.

These special constants also cause an exception with the usage of Fortran INTENT: with USE mpi_f08, the attributes INTENT(IN), INTENT(OUT), and INTENT(INOUT) are used in the Fortran interface. In most cases, INTENT(IN) is used if the C interface uses call-by-value. For all buffer arguments and for dummy arguments that may be modified and allow one of these special constants as input, an INTENT is not specified.

19.1.15 Fortran Derived Types

MPI supports passing Fortran entities of BIND(C) and SEQUENCE derived types to choice dummy arguments, provided no type component has the ALLOCATABLE or POINTER attribute.

The following code fragment shows some possible ways to send scalars or arrays of interoperable derived type in Fortran. The example assumes that all data is passed by address.

```
type, BIND(C) :: mytype
   integer :: i
  real :: x
  double precision :: d
   logical :: 1
end type mytype
type(mytype) :: foo, fooarr(5)
integer :: blocklen(4), type(4)
integer(KIND=MPI_ADDRESS_KIND) :: disp(4), base, lb, extent
call MPI_GET_ADDRESS(foo%i, disp(1), ierr)
call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
call MPI_GET_ADDRESS(foo%d, disp(3), ierr)
call MPI_GET_ADDRESS(foo%1, disp(4), ierr)
base = disp(1)
disp(1) = disp(1) - base
disp(2) = disp(2) - base
disp(3) = disp(3) - base
disp(4) = disp(4) - base
blocklen(1) = 1
```

 24

```
1
     blocklen(2) = 1
\mathbf{2}
     blocklen(3) = 1
3
     blocklen(4) = 1
4
5
     type(1) = MPI_INTEGER
6
     type(2) = MPI_REAL
\overline{7}
     type(3) = MPI_DOUBLE_PRECISION
8
     type(4) = MPI_LOGICAL
9
10
     call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr)
11
     call MPI_TYPE_COMMIT(newtype, ierr)
12
13
     call MPI_SEND(foo%i, 1, newtype, dest, tag, comm, ierr)
14
     ! or
15
     call MPI_SEND(foo, 1, newtype, dest, tag, comm, ierr)
16
     ! expects that base == address(foo%i) == address(foo)
17
18
     call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
19
     call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
20
     extent = disp(2) - disp(1)
21
     1b = 0
22
     call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
23
     call MPI_TYPE_COMMIT(newarrtype, ierr)
^{24}
     call MPI_SEND(fooarr, 5, newarrtype, dest, tag, comm, ierr)
25
```

Using the derived type variable foo instead of its first basic type element foo%i may be impossible if the MPI library implements choice buffer arguments through overloading instead of using TYPE(*), DIMENSION(...), or through a non-standardized extension such as !\$PRAGMA IGNORE_TKR; see Section 19.1.6.

To use a derived type in an array requires a correct extent of the datatype handle 31 to take care of the alignment rules applied by the compiler. These alignment rules may 32 imply that there are gaps between the components of a derived type, and also between the 33 subsuguent elements of an array of a derived type. The extent of an interoperable derived 34 type (i.e., defined with BIND(C)) and a SEQUENCE derived type with the same content may 35 be different because C and Fortran may apply different alignment rules. As recommended 36 in the advice to users in Section 5.1.6, one should add an additional fifth structure element 37 with one numerical storage unit at the end of this structure to force in most cases that 38 the array of structures is contiguous. Even with such an additional element, one should 39 keep this resizing due to the special alignment rules that can be used by the compiler for 40 structures, as also mentioned in this advice. 41

Using the extended semantics defined in TS 29113, it is also possible to use entities or derived types without either the BIND(C) or the SEQUENCE attribute as choice buffer arguments; some additional constraints must be observed, e.g., no ALLOCATABLE or POINTER type components may exist. In this case, the base address in the example must be changed to become the address of foo instead of foo%i, because the Fortran compiler may rearrange type components or add padding. Sending the structure foo should then also be performed by providing it (and not foo%i) as actual argument for MPI_Send.

19.1.16 Optimization Problems, an Overview

MPI provides operations that may be hidden from the user code and run concurrently with it, accessing the same memory as user code. Examples include the data transfer for an MPI_IRECV. The optimizer of a compiler will assume that it can recognize periods when a copy of a variable can be kept in a register without reloading from or storing to memory. When the user code is working with a register copy of some variable while the hidden operation reads or writes the memory copy, problems occur. These problems are independent of the Fortran support method; i.e., they occur with the mpi_f08 module, the mpi module, and the mpif.h include file.

This section shows four problematic usage areas (the abbreviations in parentheses are used in the table below):

- Use of nonblocking routines or persistent requests (Nonbl.).
- Use of one-sided routines (1-sided).
- Use of MPI parallel file I/O split collective operations (Split).
- Use of MPI_BOTTOM together with absolute displacements in MPI datatypes, or relative displacements between two variables in such datatypes (*Bottom*).

The following compiler optimization strategies (valid for serial code) may cause problems in MPI applications:

- Code movement and register optimization problems; see Section 19.1.17.
- Temporary data movement and temporary memory modifications; see Section 19.1.18.
- Permanent data movement (e.g., through garbage collection); see Section 19.1.19.

Table 19.2 shows the only usage areas where these optimization problems may occur.

Optimization) n	nay cause	a probl	em in
	fc	ollowing u	sage are	eas
	Nonbl.	1-sided	Split	Bottom
Code movement	yes	yes	no	yes
and register optimization				
Temporary data movement	yes	yes	yes	no
Permanent data movement	yes	yes	yes	yes

 Table 19.2: Occurrence of Fortran optimization problems in several usage areas

The solutions in the following sections are based on compromises:

- to minimize the burden for the application programmer, e.g., as shown in Sections "Solutions" through "The (Poorly Performing) Fortran VOLATILE Attribute" on pages 768–772,
- to minimize the drawbacks on compiler based optimization, and
- to minimize the requirements defined in Section 19.1.7.

 $45 \\ 46$

```
1
      19.1.17 Problems with Code Movement and Register Optimization
\mathbf{2}
      Nonblocking Operations
3
4
      If a variable is local to a Fortran subroutine (i.e., not in a module or a COMMON block), the
\mathbf{5}
      compiler will assume that it cannot be modified by a called subroutine unless it is an actual
6
      argument of the call. In the most common linkage convention, the subroutine is expected
7
      to save and restore certain registers. Thus, the optimizer will assume that a register which
8
      held a valid copy of such a variable before the call will still hold a valid copy on return.
9
10
      Example 19.1 Fortran 90 register optimization — extreme.
11
     Source
                                   compiled as
                                                                or compiled as
12
13
                                                                 REAL :: buf, b1
     REAL :: buf, b1
                                   REAL :: buf, b1
14
      call MPI_IRECV(buf,..req)
                                   call MPI_IRECV(buf,..req)
                                                                 call MPI_IRECV(buf,..req)
15
                                   register = buf
                                                                 b1 = buf
16
      call MPI_WAIT(req,..)
                                   call MPI_WAIT(req,..)
                                                                 call MPI_WAIT(req,..)
17
      b1 = buf
                                   b1 = register
18
19
          Example 19.1 shows extreme, but allowed, possibilities. MPI_WAIT on a concurrent
20
      thread modifies buf between the invocation of MPI_IRECV and the completion of MPI_WAIT.
21
      But the compiler cannot see any possibility that buf can be changed after MPI_IRECV has
22
      returned, and may schedule the load of buf earlier than typed in the source. The compiler
23
      has no reason to avoid using a register to hold buf across the call to MPI_WAIT. It also may
^{24}
      reorder the instructions as illustrated in the rightmost column.
25
26
      Example 19.2 Similar example with MPI_ISEND
27
28
      Source
                                   compiled as
                                                                with a possible MPI-internal
                                                                 execution sequence
29
30
                                   REAL :: buf, copy
     REAL :: buf, copy
                                                                 REAL :: buf, copy
^{31}
      buf = val
                                   buf = val
                                                                 buf = val
32
                                                                 addr = &buf
      call MPI_ISEND(buf, .. req)
                                   call MPI_ISEND(buf,..req)
33
     copy = buf
                                   copy= buf
                                                                 copy = buf
34
                                   buf = val_overwrite
                                                                 buf = val_overwrite
     call MPI_WAIT(req,..)
                                   call MPI_WAIT(req,..)
                                                                 call send(*addr) ! within
35
                                                                                    ! MPI_WAIT
36
     buf = val_overwrite
37
38
39
          Due to valid compiler code movement optimizations in Example 19.2, the content of
```

⁵³ Due to valid compiler code movement optimizations in Example 19.2, the content of ⁴⁰ buf may already have been overwritten by the compiler when the content of buf is sent. ⁴¹ The code movement is permitted because the compiler cannot detect a possible access to ⁴² buf in MPI_WAIT (or in a second thread between the start of MPI_ISEND and the end of ⁴³ MPI_WAIT).

Such register optimization is based on moving code; here, the access to buf was moved
 from after MPI_WAIT to before MPI_WAIT. Note that code movement may also occur across
 subroutine boundaries when subroutines or functions are inlined.

This register optimization/code movement problem for nonblocking operations does not occur with MPI parallel file I/O split collective operations, because in the ..._BEGIN

and ..._END calls, the same buffer has to be provided as an actual argument. The register optimization / code movement problem for MPI_BOTTOM and derived MPI datatypes may occur in each blocking and nonblocking communication call, as well as in each parallel file I/O operation.

Persistent Operations

With persistent requests, the buffer argument is hidden from the MPI_START and MPI_STARTALL calls, i.e., the Fortran compiler may move buffer accesses across the MPI_START or MPI_STARTALL call, similar to the MPI_WAIT call as described in the Nonblocking Operations subsection in Section 19.1.17.

One-sided Communication

An example with instruction reordering due to register optimization can be found in Section 12.7.4.

MPI_BOTTOM and Combining Independent Variables in Datatypes

This section is only relevant if the MPI program uses a buffer argument to an MPI_SEND, MPI_RECV, etc., that hides the actual variables involved in the communication. MPI_BOTTOM with an MPI_Datatype containing *absolute addresses* is one example. Creating a datatype which uses one variable as an anchor and brings along others by using MPI_GET_ADDRESS to determine their offsets from the anchor is another. The anchor variable would be the only one referenced in the call. Also attention must be paid if MPI operations are used that run in parallel with the user's application.

Example 19.3 shows what Fortran compilers are allowed to do.

Example 19.3	Fortran	90 register	optimization.
--------------	---------	--------------	---------------

This source	can be compiled as:
call MPI_GET_ADDRESS(buf,bufaddr, ierror)	<pre>call MPI_GET_ADDRESS(buf,)</pre>
<pre>call MPI_TYPE_CREATE_STRUCT(1,1,</pre>	call MPI_TYPE_CREATE_STRUCT()
<pre>call MPI_TYPE_COMMIT(type,ierror)</pre>	<pre>call MPI_TYPE_COMMIT()</pre>
val_old = buf	register = buf val_old = register
<pre>call MPI_RECV(MPI_BOTTOM,1,type,) val_new = buf</pre>	<pre>call MPI_RECV(MPI_BOTTOM,) val_new = register</pre>

In Example 19.3, the compiler does not invalidate the register because it cannot see that MPI_RECV changes the value of buf. The access to buf is hidden by the use of MPI_GET_ADDRESS and MPI_BOTTOM.

In Example 19.4, several successive assignments to the same variable **buf** can be combined in a way such that only the last assignment is executed. "Successive" means that no interfering load access to this variable occurs between the assignments. The compiler

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

1 Example 19.4 Similar example with MPI_SEND $\mathbf{2}$ 3 This source ... can be compiled as: 4 ! buf contains val_old ! buf contains val_old 5buf = val_new 6 call MPI_SEND(MPI_BOTTOM,1,type,...) call MPI_SEND(...) 7 ! with buf as a displacement in type ! i.e. val_old is sent 8 1 9 ! buf=val_new is moved to here 10 ! and detected as dead code 11 ! and therefore removed 121 13 buf = val_overwrite buf = val_overwrite 14 1516cannot detect that the call to MPI_SEND statement is interfering because the load access 17to buf is hidden by the usage of MPI_BOTTOM. 18 19 Solutions 2021The following sections show in detail how the problems with code movement and register 22 optimization can be portably solved. Application writers can partially or fully avoid these 23compiler optimization problems by using one or more of the special Fortran declarations 24with the send and receive buffers used in nonblocking operations, or in operations in which 25MPI_BOTTOM is used, or if datatype handles that combine several variables are used: 26• Use of the Fortran ASYNCHRONOUS attribute. 2728• Use of the helper routine MPI_F_SYNC_REG, or an equivalent user-written dummy 29 routine. 30 31• Declare the buffer as a Fortran module variable or within a Fortran common block. 32 • Use of the Fortran VOLATILE attribute. 33 34 Each of these methods solves the problems of code movement and register optimization, 35 but may incur various degrees of performance impact, and may not be usable in every 36 application context. These methods may not be guaranteed by the Fortran standard, but 37 they must be guaranteed by a MPI-3.0 (and later) compliant MPI library and associated 38 compiler suite according to the requirements listed in Section 19.1.7. The performance 39 impact of using MPI_F_SYNC_REG is expected to be low, that of using module variables 40 or the ASYNCHRONOUS attribute is expected to be low to medium, and that of using the 41 VOLATILE attribute is expected to be high or very high. Note that there is one attribute 42that cannot be used for this purpose: the Fortran TARGET attribute does not solve code 43 movement problems in MPI applications. 44 45

46 The Fortran ASYNCHRONOUS Attribute

⁴⁷ Declaring an actual buffer argument with the ASYNCHRONOUS Fortran attribute in a scoping
 ⁴⁸ unit (or BLOCK) informs the compiler that any statement in the scoping unit may be executed

while the buffer is affected by a pending asynchronous Fortran input/output operation (since $\mathbf{2}$ Fortran 2003) or by an asynchronous communication (TS 29113 extension). Without the 3 extensions specified in TS 29113, a Fortran compiler may totally ignore this attribute if the 4 Fortran compiler implements asynchronous Fortran input/output operations with blocking $\mathbf{5}$ I/O. The ASYNCHRONOUS attribute protects the buffer accesses from optimizations through code movements across routine calls, and the buffer itself from temporary and permanent 6 7 data movements. If the choice buffer dummy argument of a nonblocking MPI routine is declared with ASYNCHRONOUS (which is mandatory for the mpi_f08 module, with allowable 8 9 exceptions listed in Section 19.1.6), then the compiler has to guarantee call by reference 10 and should report a compile-time error if call by reference is impossible, e.g., if vector subscripts are used. The MPI_ASYNC_PROTECTS_NONBLOCKING is set to .TRUE. if both 11the protection of the actual buffer argument through ASYNCHRONOUS according to the TS 1213 29113 extension and the declaration of the dummy argument with ASYNCHRONOUS in the Fortran support method is guaranteed for all nonblocking routines, otherwise it is set to 1415.FALSE..

The ASYNCHRONOUS attribute has some restrictions. Section 5.4.2 of the TS 29113 specifies:

"Asynchronous communication for a Fortran variable occurs through the action of procedures defined by means other than Fortran. It is initiated by execution of an asynchronous communication initiation procedure and completed by execution of an asynchronous communication completion procedure. Between the execution of the initiation and completion procedures, any variable of which any part is associated with any part of the asynchronous communication variable is a pending communication affector. Whether a procedure is an asynchronous communication initiation or completion procedure is processor dependent.

Asynchronous communication is either input communication or output communication. For input communication, a pending communication affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed. For output communication, a pending communication affector shall not be redefined, become undefined, or have its pointer association status changed."

In Example 19.5 Case (a) on page 775, the read accesses to b within function(b(i-1), b(i), b(i+1)) cannot be moved by compiler optimizations to before the wait call because b was declared as ASYNCHRONOUS. Note that only the elements 0, 1, 100, and 101 of b are involved in asynchronous communication but by definition, the total variable **b** is the pending communication affector and is usable for input and output asynchronous communication between the MPI_I... routines and MPI_Waitall. Case (a) works fine because the read accesses to b occur after the communication has completed.

In Case (b), the read accesses to b(1:100) in the loop i=2,99 are read accesses to 42a pending communication affector while input communication (i.e., the two MPI_Irecv 43 calls) is pending. This is a contradiction to the rule that for input communication, a 44 pending communication affector shall not be referenced. The problem can be solved by using 45separate variables for the halos and the inner array, or by splitting a common array into 46 disjoint subarrays which are passed through different dummy arguments into a subroutine, 47as shown in Example 19.9.

Unofficial Draft for Comment Only

1

1617

18

19

20

21

22

23

 24

25

26

2728

29

30 31

32 33

34

35

36

37

38

39

40

41

1 2 3 4 5 6 7	If one does not overlap communication and computation on the same variable, then all optimization problems can be solved through the ASYNCHRONOUS attribute. The problems with MPI_BOTTOM, as shown in Example 19.3 and Example 19.4, can also be solved by declaring the buffer buf with the ASYNCHRONOUS attribute. In some MPI routines, a buffer dummy argument is defined as ASYNCHRONOUS to guarantee passing by reference, provided that the actual argument is also defined as ASYNCHRONOUS.
8	Calling MPI_F_SYNC_REG
9 10 11 12 13	The compiler may be prevented from moving a reference to a buffer across a call to an MPI subroutine by surrounding the call by calls to an external subroutine with the buffer as an actual argument. The MPI library provides the MPI_F_SYNC_REG routine for this purpose; see Section 19.1.8.
14 15 16	• The problems illustrated by the Examples 19.1 and 19.2 can be solved by calling MPI_F_SYNC_REG(buf) once immediately after MPI_WAIT.
17	Example 19.1 Example 19.2
18	can be solved with can be solved with
19	call MPI_IRECV(buf,req) buf = val
20	<pre>call MPI_ISEND(buf,req)</pre>
21 22	copy = buf
22	call MPI_WAIT(req,) call MPI_WAIT(req,)
24	call MPI_F_SYNC_REG(buf) call MPI_F_SYNC_REG(buf)
25	b1 = buf buf = val_overwrite
26	The call to MPI_F_SYNC_REG(buf) prevents moving the last line before the
27	MPI_WAIT call. Further calls to MPI_F_SYNC_REG(buf) are not needed because it
28	is still correct if the additional read access copy=buf is moved below MPI_WAIT and
29	before buf=val_overwrite.
30	
31	• The problems illustrated by the Examples 19.3 and 19.4 can be solved with two
32	additional MPI_F_SYNC_REG(buf) statements; one directly before MPI_RECV/
33	MPI_SEND, and one directly after this communication operation.
$\frac{34}{35}$	Example 19.3 Example 19.4
36	can be solved with can be solved with
37	call MPI_F_SYNC_REG(buf) call MPI_F_SYNC_REG(buf)
38	call MPI_RECV(MPI_BOTTOM,) call MPI_SEND(MPI_BOTTOM,)
39	call MPI_F_SYNC_REG(buf) call MPI_F_SYNC_REG(buf)
40	
41	The first call to MPI_F_SYNC_REG(buf) is needed to finish all load and store refer-
42	ences to buf prior to MPI_RECV/MPI_SEND; the second call is needed to assure that
43	any subsequent access to buf is not moved before MPI_RECV/MPI_SEND.
44	• In the example in Section 12.7.4, two asynchronous accesses must be protected: in
45	Process 1, the access to bbbb must be protected similar to Example 19.1, i.e., a call to
46	$MPI_FSYNC_REG(bbbb)$ is needed after the second MPI_WIN_FENCE to guarantee
47	that further accesses to $bbbb$ are not moved ahead of the call to $MPI_WIN_FENCE.$ In
48	Process 2, both calls to MPI_WIN_FENCE together act as a communication call with

MPI_BOTTOM as the buffer. That is, before the first fence and after the second fence, a call to MPI_F_SYNC_REG(buff) is needed to guarantee that accesses to buff are not moved after or ahead of the calls to MPI_WIN_FENCE. Using MPI_GET instead of MPI_PUT, the same calls to MPI_F_SYNC_REG are necessary.

Source of 2	Process 1	Source of Process 2
bbbb = 777		buff = 999
		call MPI_F_SYNC_REG(buff)
call MPI_W	IN_FENCE	call MPI_WIN_FENCE
call MPI_P	UT(bbbb	
into buff	of process 2)	
call MPI_W	IN_FENCE	call MPI_WIN_FENCE
call MPI_F	_SYNC_REG(bbbb)	call MPI_F_SYNC_REG(buff)
		ccc = buff

• The temporary memory modification problem, i.e., Example 19.6, can **not** be solved with this method.

A User Defined Routine Instead of MPI_F_SYNC_REG

Instead of MPI_F_SYNC_REG, one can also use a user defined external subroutine, which is separately compiled:

```
subroutine DD(buf)
integer buf
end
```

Note that if the intent is declared in an explicit interface for the external subroutine, it must be OUT or INOUT. The subroutine itself may have an empty body, but the compiler does not know this and has to assume that the buffer may be altered. For example, a call to MPI_RECV with MPI_BOTTOM as buffer might be replaced by

call	DD(buf)
call	<pre>MPI_RECV(MPI_BOTTOM,)</pre>
call	DD(buf)

Such a user-defined routine was introduced in MPI-2.0 and is still included here to document such usage in existing application programs although new applications should prefer MPI_F_SYNC_REG or one of the other possibilities. In an existing application, calls to such a user-written routine should be substituted by a call to MPI_F_SYNC_REG because the user-written routine may not be implemented in accordance with the rules specified in Section 19.1.7.

Module Variables and COMMON Blocks

An alternative to the previously mentioned methods is to put the buffer or variable into a ⁴⁶ module or a common block and access it through a USE or COMMON statement in each scope ⁴⁷ where it is referenced, defined or appears as an actual argument in a call to an MPI routine. ⁴⁸

Unofficial Draft for Comment Only

 $\mathbf{2}$

The compiler will then have to assume that the MPI procedure may alter the buffer or
 variable, provided that the compiler cannot infer that the MPI procedure does not reference
 the module or common block.

- This method solves problems of instruction reordering, code movement, and register optimization related to nonblocking and one-sided communication, or related to the usage of MPI_BOTTOM and derived datatype handles.
- Unfortunately, this method does **not** solve problems caused by asynchronous accesses between the start and end of a nonblocking or one-sided communication. Specifically, problems caused by temporary memory modifications are not solved.
- 11 12

13

4

5

6

7 8

9

10

The (Poorly Performing) Fortran VOLATILE Attribute

The VOLATILE attribute gives the buffer or variable the properties needed to avoid register optimization or code movement problems, but it may inhibit optimization of any code containing references or definitions of the buffer or variable. On many modern systems, the performance impact will be large because not only register, but also cache optimizations will not be applied. Therefore, use of the VOLATILE attribute to enforce correct execution of MPI programs is discouraged.

20 21

22

The Fortran TARGET Attribute

The TARGET attribute does not solve the code movement problem because it is not specified for the choice buffer dummy arguments of nonblocking routines. If the compiler detects that the application program specifies the TARGET attribute for an actual buffer argument used in the call to a nonblocking routine, the compiler may ignore this attribute if no pointer reference to this buffer exists.

- 28
- 29 30

Rationale. The Fortran standardization body decided to extend the ASYNCHRONOUS attribute within the TS 29113 to protect buffers in nonblocking calls from all kinds of optimization, instead of extending the TARGET attribute. (*End of rationale.*)

31 32 33

19.1.18 Temporary Data Movement and Temporary Memory Modification

The compiler is allowed to temporarily modify data in memory. Normally, this problem may occur only when overlapping communication and computation, as in Example 19.5, Case (b) on page 775. Example 19.6 also shows a possibility that could be problematic.

In the compiler-generated, possible optimization in Example 19.7, buf(100,100) from Example 19.6 is equivalenced with the 1-dimensional array buf_1dim(10000). The nonblocking receive may asynchronously receive the data in the boundary buf(1,1:100) while the fused loop is temporarily using this part of the buffer. When the tmp data is written back to buf, the previous data of buf(1,1:100) is restored and the received data is lost. The principle behind this optimization is that the receive buffer data buf(1,1:100) was temporarily moved to tmp.

Example 19.8 shows a second possible optimization. The whole array is temporarily
 moved to local_buf.

When storing local_buf back to the original location buf, then this implies overwriting the section of buf that serves as a receive buffer in the nonblocking MPI call, i.e., this 8 storing back of local_buf is therefore likely to interfere with asynchronously received data in buf(1,1:100).

Note that this problem may also occur:

- With the local buffer at the origin process, between an RMA communication call and the ensuing synchronization call; see Chapter 12.
- With the window buffer at the target process between two ensuing RMA synchronization calls.
- With the local buffer in MPI parallel file I/O split collective operations between the ..._BEGIN and ..._END calls; see Section 14.4.5.

As already mentioned in subsection *The Fortran ASYNCHRONOUS attribute* on page 768 of Section 19.1.17, the ASYNCHRONOUS attribute can prevent compiler optimization with temporary data movement, but only if the receive buffer and the local references are separated into different variables, as shown in Example 19.9 and in Example 19.10.

Note also that the methods

- calling MPI_F_SYNC_REG (or such a user-defined routine),
- using module variables and COMMON blocks, and
- the TARGET attribute

cannot be used to prevent such temporary data movement. These methods influence compiler optimization when library routines are called. They cannot prevent the optimizations of the code fragments shown in Example 19.6 and 19.7.

Note also that compiler optimization with temporary data movement should **not** be prevented by declaring **buf** as **VOLATILE** because the **VOLATILE** implies that all accesses to any storage unit (word) of **buf** must be directly done in the main memory exactly in the sequence defined by the application program. The **VOLATILE** attribute prevents all register and cache optimizations. Therefore, **VOLATILE** may cause a huge performance degradation.

Instead of solving the problem, it is better to **prevent** the problem: when overlapping communication and computation, the nonblocking communication (or nonblocking or split collective I/O) and the computation should be executed **on different variables**, and the communication should be *protected* with the ASYNCHRONOUS attribute. In this case, the temporary memory modifications are done only on the variables used in the computation and cannot have any side effect on the data used in the nonblocking MPI operations.

Rationale. This is a strong restriction for application programs. To weaken this restriction, a new or modified asynchronous feature in the Fortran language would be necessary: an asynchronous attribute that can be used on parts of an array and together with asynchronous operations outside the scope of Fortran. If such a feature becomes available in a future edition of the Fortran standard, then this restriction also may be weakened in a later version of the MPI standard. (*End of rationale.*)

In Example 19.9 (which is a solution for the problem shown in Example 19.5 and in Example 19.10 (which is a solution for the problem shown in Example 19.8), the array is split into inner and halo part and both disjoint parts are passed to a subroutine separated_sections. This routine overlaps the receiving of the halo data and the calculations on the inner part of the array. In a second step, the whole array is used to do the

Unofficial Draft for Comment Only

2

3

4

5

6 7

8

9

10

11 12

13

14

15

16

17 18

19

20 21

22

23

 24

25

26

27

28

29

30

31

32

33

34

35

36 37

38

39

40

41

42

1 calculation on the elements where inner+halo is needed. Note that the halo and the inner $\mathbf{2}$ area are strided arrays. Those can be used in nonblocking communication only with a TS 3 29113 based MPI library.

4 5

6

7

19.1.19 Permanent Data Movement

A Fortran compiler may implement permanent data movement during the execution of a Fortran program. This would require that pointers to such data are appropriately updated. 8 An implementation with automatic garbage collection is one use case. Such permanent data 9 movement is in conflict with MPI in several areas: 10

11 12

13

14

15

21

- MPI datatype handles with absolute addresses in combination with MPI_BOTTOM.
- All nonblocking MPI operations if the internally used pointers to the buffers are not updated by the Fortran runtime, or if within an MPI process, the data movement is executed in parallel with the MPI operation.

16This problem can be also solved by using the ASYNCHRONOUS attribute for such buffers. 17This MPI standard requires that the problems with permanent data movement do not 18 occur by imposing suitable restrictions on the MPI library together with the compiler used; 19 see Section 19.1.7. 20

19.1.20 Comparison with C 22

23In C, subroutines which modify variables that are not in the argument list will not cause 24 register optimization problems. This is because taking pointers to storage objects by using 25the & operator and later referencing the objects by indirection on the pointer is an integral 26part of the language. A C compiler understands the implications, so that the problem should 27not occur, in general. However, some compilers do offer optional aggressive optimization 28levels which may not be safe. Problems due to temporary memory modifications can also 29occur in C. As above, the best advice is to avoid the problem: use different variables for 30 buffers in nonblocking MPI operations and computation that is executed while a nonblocking 31 operation is pending. 32

- 33
- 34

Example 19.5 Protecting nonblocking communication with the ASYNCHRONOUS attribute.

```
USE mpi_f08
REAL, ASYNCHRONOUS :: b(0:101) ! elements 0 and 101 are halo cells
                                ! elements 1 and 100 are newly computed
REAL :: bnew(0:101)
TYPE(MPI_Request) :: req(4)
INTEGER :: left, right, i
CALL MPI_Cart_shift(...,left,right,...)
CALL MPI_Irecv(b( 0), ..., left, ..., req(1), ...)
CALL MPI_Irecv(b(101), ..., right, ..., req(2), ...)
CALL MPI_Isend(b( 1), ..., left, ..., req(3), ...)
CALL MPI_Isend(b(100), ..., right, ..., req(4), ...)
#ifdef WITHOUT_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (a)
  CALL MPI_Waitall(4, req, ...)
  DO i=1,100 ! compute all new local data
    bnew(i) = function(b(i-1), b(i), b(i+1))
  END DO
                                                                                 21
#endif
                                                                                 22
                                                                                 23
#ifdef WITH_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (b)
  DO i=2,99 ! compute only elements for which halo data is not needed
    bnew(i) = function(b(i-1), b(i), b(i+1))
  END DO
  CALL MPI_Waital1(4, req, ...)
  i=1 ! compute leftmost element
                                                                                 30
    bnew(i) = function(b(i-1), b(i), b(i+1))
  i=100 ! compute rightmost element
    bnew(i) = function(b(i-1), b(i), b(i+1))
#endif
                                                                                 34
                                                                                 35
                                                                                 36
Example 19.6 Overlapping Communication and Computation.
                                                                                 37
USE mpi_f08
REAL :: buf(100,100)
CALL MPI_Irecv(buf(1,1:100),..., req,...)
DO j=1,100
  DO i=2,100
    buf(i,j)=...
  END DO
END DO
CALL MPI_Wait(req,...)
```

2 3 4

5

6

9

10

11

12

13

1415

16

17

18

19

20

 24

25

26

27

28

29

31

32

33

38

39

40

41

42

43

44

45

46

```
1
     Example 19.7 The compiler may substitute the nested loops through loop fusion.
\mathbf{2}
3
     REAL :: buf(100,100), buf_1dim(10000)
4
     EQUIVALENCE (buf(1,1), buf_1dim(1))
\mathbf{5}
     CALL MPI_Irecv(buf(1,1:100),..., req,...)
6
     tmp(1:100) = buf(1,1:100)
\overline{7}
     DO j=1,10000
8
       buf_1dim(h)=...
9
     END DO
10
     buf(1,1:100) = tmp(1:100)
11
     CALL MPI_Wait(req,...)
12
13
14
15
16
17
18
19
20
21
22
23
^{24}
25
26
27
     Example 19.8 Another optimization is based on the usage of a separate memory storage
28
     area, e.g., in a GPU.
29
30
     REAL :: buf(100,100), local_buf(100,100)
31
     CALL MPI_Irecv(buf(1,1:100),..., req,...)
32
     local_buf = buf
33
     DO j=1,100
34
       DO i=2,100
35
          local_buf(i,j)=...
36
        END DO
37
     END DO
38
     buf = local_buf ! may overwrite asynchronously received
39
                      ! data in buf(1,1:100)
40
     CALL MPI_Wait(req,...)
41
42
43
44
45
46
47
48
```

Example 19.9 Using separated variables for overlapping communication and computation to allow the protection of nonblocking communication with the ASYNCHRONOUS attribute.

```
4
USE mpi_f08
                                                                                    5
REAL :: b(0:101)
                      ! elements 0 and 101 are halo cells
                                                                                    6
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
                                                                                    7
INTEGER :: i
CALL separated_sections(b(0), b(1:100), b(101), bnew(0:101))
                                                                                    9
i=1 ! compute leftmost element
                                                                                    10
  bnew(i) = function(b(i-1), b(i), b(i+1))
                                                                                    11
i=100 ! compute rightmost element
                                                                                    12
  bnew(i) = function(b(i-1), b(i), b(i+1))
                                                                                    13
END
                                                                                    14
                                                                                    15
SUBROUTINE separated_sections(b_lefthalo, b_inner, b_righthalo, bnew)
                                                                                    16
USE mpi_f08
                                                                                    17
REAL, ASYNCHRONOUS :: b_lefthalo(0:0), b_inner(1:100), b_righthalo(101:101)
                                                                                    18
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
                                                                                    19
TYPE(MPI_Request) :: req(4)
                                                                                    20
INTEGER :: left, right, i
                                                                                    21
CALL MPI_Cart_shift(...,left, right,...)
                                                                                    22
CALL MPI_Irecv(b_lefthalo ( 0), ..., left, ..., req(1), ...)
                                                                                    23
CALL MPI_Irecv(b_righthalo(101), ..., right, ..., req(2), ...)
                                                                                    ^{24}
! b_lefthalo and b_righthalo is written asynchronously.
                                                                                    25
! There is no other concurrent access to b_lefthalo and b_righthalo.
                                                                                    26
CALL MPI_Isend(b_inner( 1),
                                   ..., left, ..., req(3), ...)
                                                                                    27
CALL MPI_Isend(b_inner(100),
                                   ..., right, ..., req(4), ...)
                                                                                    28
                                                                                    29
DO i=2,99 ! compute only elements for which halo data is not needed
                                                                                    30
  bnew(i) = function(b_inner(i-1), b_inner(i), b_inner(i+1))
                                                                                    31
  ! b_inner is read and sent at the same time.
                                                                                    32
  ! This is allowed based on the rules for ASYNCHRONOUS.
                                                                                    33
END DO
                                                                                    34
CALL MPI_Waitall(4, req,...)
                                                                                    35
END SUBROUTINE
                                                                                    36
                                                                                    37
                                                                                    38
                                                                                    39
                                                                                    40
                                                                                    41
                                                                                    42
                                                                                    43
                                                                                    44
                                                                                    45
                                                                                    46
                                                                                    47
                                                                                    48
```

1

```
1
     Example 19.10 Protecting GPU optimizations with the ASYNCHRONOUS attribute.
\mathbf{2}
3
     USE mpi_f08
4
     REAL :: buf(100,100)
\mathbf{5}
     CALL separated_sections(buf(1:1,1:100), buf(2:100,1:100))
6
     END
\overline{7}
8
     SUBROUTINE separated_sections(buf_halo, buf_inner)
9
     REAL, ASYNCHRONOUS :: buf_halo(1:1,1:100)
10
     REAL :: buf_inner(2:100,1:100)
11
     REAL :: local_buf(2:100,100)
12
13
     CALL MPI_Irecv(buf_halo(1,1:100),..., req,...)
14
     local_buf = buf_inner
15
     DO j=1,100
16
       DO i=2,100
17
          local_buf(i,j)=...
18
       END DO
19
     END DO
20
     buf_inner = local_buf ! buf_halo is not touched!!!
21
^{22}
     CALL MPI_Wait(req,...)
23
24
25
26
27
28
29
30
^{31}
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

19.2 Language Interoperability

19.2.1 Introduction

It is not uncommon for library developers to use one language to develop an application library that may be called by an application program written in a different language. MPI currently supports ISO (previously ANSI) C and Fortran bindings. It should be possible for applications in any of the supported languages to call MPI-related functions in another language.

Moreover, MPI allows the development of client-server code, with MPI communication used between a parallel client and a parallel server. It should be possible to code the server in one language and the clients in another language. To do so, communications should be possible between applications written in different languages.

There are several issues that need to be addressed in order to achieve interoperability.

Initialization We need to specify how the MPI environment is initialized for all languages.

- Interlanguage passing of MPI opaque objects We need to specify how MPI object handles are passed between languages. We also need to specify what happens when an MPI object is accessed in one language, to retrieve information (e.g., attributes) set in another language.
- **Interlanguage communication** We need to specify how messages sent in one language can be received in another language.

It is highly desirable that the solution for interlanguage interoperability be extensible to new languages, should MPI bindings be defined for such languages.

19.2.2 Assumptions

We assume that conventions exist for programs written in one language to call routines written in another language. These conventions specify how to link routines in different languages into one program, how to call functions in a different language, how to pass arguments between languages, and the correspondence between basic data types in different languages. In general, these conventions will be implementation dependent. Furthermore, not every basic datatype may have a matching type in other languages. For example, C character strings may not be compatible with Fortran CHARACTER variables. However, we assume that a Fortran INTEGER, as well as a (sequence associated) Fortran array of INTEGERs, can be passed to a C program. We also assume that Fortran and C have addresssized integers. This does not mean that the default-size integers are the same size as default-sized pointers, but only that there is some way to hold (and pass) a C address in a Fortran integer. It is also assumed that INTEGER(KIND=MPI_OFFSET_KIND) can be passed from Fortran to C as MPI_Offset.

19.2.3 Initialization

A call to MPI_INIT or MPI_INIT_THREAD, from any language, initializes MPI for execution in all languages.

Advice to users. Certain implementations use the (inout) argc, argv arguments of the C version of MPI_INIT in order to propagate values for argc and argv to all

Unofficial Draft for Comment Only

1 2

3 4

5

6

 $\overline{7}$

8

9

10

11

12

13

14

15 16

17

18

19

20 21

22

23

 24

25 26

27 28

29

30

 31

32

33

34

35

36

37

38

39

40

41 42

43 44

45

 $46 \\ 47$

1 executing processes. Use of the Fortran version of MPI_INIT to initialize MPI may 2 result in a loss of this ability. (End of advice to users.) 3 The function MPI_INITIALIZED returns the same answer in all languages. 4 The function MPI_FINALIZE finalizes the MPI environments for all languages. 5The function MPI_FINALIZED returns the same answer in all languages. 6 The function MPI_ABORT kills processes, irrespective of the language used by the 7 caller or by the processes killed. 8 9 The MPI environment is initialized in the same manner for all languages by 10 MPI_INIT. E.g., MPI_COMM_WORLD carries the same information regardless of language: 11 same processes, same environmental attributes, same error handlers. Information can be added to info objects in one language and retrieved in another. 1213 Advice to users. The use of several languages in one MPI program may require the 14use of special options at compile and/or link time. (End of advice to users.) 1516Advice to implementors. Implementations may selectively link language specific MPI 17 libraries only to codes that need them, so as not to increase the size of binaries for codes 18 that use only one language. The MPI initialization code need perform initialization for 19 a language only if that language library is loaded. (End of advice to implementors.) 202119.2.4 Transfer of Handles 22 23Handles are passed between Fortran and C by using an explicit C wrapper to convert Fortran 24 handles to C handles. There is no direct access to C handles in Fortran. 25The type definition MPI_Fint is provided in C for an integer of the size that matches a 26Fortran INTEGER; usually, MPI_Fint will be equivalent to int. With the Fortran mpi module 27or the mpif.h include file, a Fortran handle is a Fortran INTEGER value that can be used in 28the following conversion functions. With the Fortran mpi_f08 module, a Fortran handle is a 29 BIND(C) derived type that contains an INTEGER component named MPI_VAL. This INTEGER 30 value can be used in the following conversion functions. 31 The following functions are provided in C to convert from a Fortran communicator 32 handle (which is an integer) to a C communicator handle, and vice versa. See also Sec-33 tion 2.6.4. 34 C binding 35 MPI_Comm MPI_Comm_f2c(MPI_Fint comm) 36 If comm is a valid Fortran handle to a communicator, then MPI_Comm_f2c returns a 37 valid C handle to that same communicator; if $comm = MPI_COMM_NULL$ (Fortran value), 38 then MPI_Comm_f2c returns a null C handle; if comm is an invalid Fortran handle, then 39 MPI_Comm_f2c returns an invalid C handle. 40 MPI_Fint MPI_Comm_c2f(MPI_Comm comm) 41 42The function MPI_Comm_c2f translates a C communicator handle into a Fortran handle 43 to the same communicator; it maps a null handle into a null handle and an invalid handle 44 into an invalid handle. 45 Similar functions are provided for the other types of opaque objects. 46 MPI_Datatype MPI_Type_f2c(MPI_Fint datatype) 47

⁴⁸ MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)

MPI_Group MPI_Group_f2c(MPI_Fint group)	1
MPI_Fint MPI_Group_c2f(MPI_Group group)	2 3
MPI_Request MPI_Request_f2c(MPI_Fint request)	4
MPI_Fint MPI_Request_c2f(MPI_Request request)	5
· · ·	6 7
MPI_File MPI_File_f2c(MPI_Fint file)	8
MPI_Fint MPI_File_c2f(MPI_File file)	9
MPI_Win MPI_Win_f2c(MPI_Fint win)	10 11
MPI_Fint MPI_Win_c2f(MPI_Win win)	12
MPI_Op MPI_Op_f2c(MPI_Fint op)	13
MPI_Fint MPI_Op_c2f(MPI_Op op)	14 15
	16
MPI_Info MPI_Info_f2c(MPI_Fint info)	17
MPI_Fint MPI_Info_c2f(MPI_Info info)	18
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)	19 20
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)	21
MPI_Message MPI_Message_f2c(MPI_Fint message)	22 23
MPI_Fint MPI_Message_c2f(MPI_Message message)	23
MPI_Session MPI_Session_f2c(MPI_Fint session)	25
MPI_Fint MPI_Session_c2f(MPI_Session session)	26 27
	28
Example 19.11 The example below illustrates how the Fortran MPI function	29
MPI_TYPE_COMMIT can be implemented by wrapping the C MPI function	30 31
MPI_Type_commit with a C wrapper to do handle conversions. In this example a Fortran-C	32
interface is assumed where a Fortran function is all upper case when referred to from C and	33
arguments are passed by addresses.	34
! FORTRAN PROCEDURE	35
SUBROUTINE MPI_TYPE_COMMIT(DATATYPE, IERR)	36
INTEGER :: DATATYPE, IERR	37
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)	38
RETURN	39
END	40 41
/* C uropper */	42
/* C wrapper */	43
<pre>void MPI_X_TYPE_COMMIT(MPI_Fint *f_handle, MPI_Fint *ierr)</pre>	44
{	45
MPI_Datatype datatype;	46
	47
<pre>datatype = MPI_Type_f2c(*f_handle);</pre>	48

```
datatype = MPI_Type_f2c(*f_handle);
```

```
1
         *ierr = (MPI_Fint)MPI_Type_commit(&datatype);
\mathbf{2}
         *f_handle = MPI_Type_c2f(datatype);
3
         return;
4
     }
5
          The same approach can be used for all other MPI functions. The call to MPI_XXX_f2c
6
      (resp. MPI_XXX_c2f) can be omitted when the handle is an OUT (resp. IN) argument, rather
7
      than INOUT.
8
9
                        The design here provides a convenient solution for the prevalent case,
           Rationale.
10
           where a C wrapper is used to allow Fortran code to call a C library, or C code to
11
           call a Fortran library. The use of C wrappers is much more likely than the use of
12
           Fortran wrappers, because it is much more likely that a variable of type INTEGER can
13
           be passed to C, than a C handle can be passed to Fortran.
14
15
           Returning the converted value as a function value rather than through the argument
16
           list allows the generation of efficient inlined code when these functions are simple
17
           (e.g., the identity). The conversion function in the wrapper does not catch an invalid
18
           handle argument. Instead, an invalid handle is passed below to the library function,
19
           which, presumably, checks its input arguments. (End of rationale.)
20
21
      19.2.5 Status
22
      The following two procedures are provided in C to convert from a Fortran (with the mpi
23
      module or mpif.h) status (which is an array of integers) to a C status (which is a structure),
24
      and vice versa. The conversion occurs on all the information in status, including that which
25
      is hidden. That is, no status information is lost in the conversion.
26
27
      int MPI Status f2c(const MPI Fint *f status, MPI Status *c_status)
28
          If f_status is a valid Fortran status, but not the Fortran value of MPI_STATUS_IGNORE
29
      or MPI_STATUSES_IGNORE, then MPI_Status_f2c returns in c_status a valid C status with
30
      the same content. If f_status is the Fortran value of MPI_STATUS_IGNORE or
^{31}
      MPI_STATUSES_IGNORE, or if f_status is not a valid Fortran status, then the call is erroneous.
32
          In C, such an f_status array can be defined with MPI_Fint f_status[
33
      MPI_F_STATUS_SIZE]. Within this array, one can use in C the indexes MPI_F_SOURCE,
34
      MPI_F_TAG, and MPI_F_ERROR, to access the same elements as in Fortran with MPI_SOURCE,
35
      MPI_TAG and MPI_ERROR. The C indexes are 1 less than the corresponding indexes in
36
      Fortran due to the different default array start indexes in both languages.
37
          The C status has the same source, tag and error code values as the Fortran status,
38
     and returns the same answers when queried for count, elements, and cancellation. The
39
      conversion function may be called with a Fortran status argument that has an undefined
40
      error field, in which case the value of the error field in the C status argument is undefined.
41
          Two global variables of type MPI_Fint*, MPI_F_STATUS_IGNORE and
42
      MPI_F_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether
43
      f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE defined in
44
      the mpi module or mpif.h. These are global variables, not C constant expressions and
45
      cannot be used in places where C requires constant expressions. Their value is defined only
46
      between the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code.
47
          To do the conversion in the other direction, we have the following:
48
```

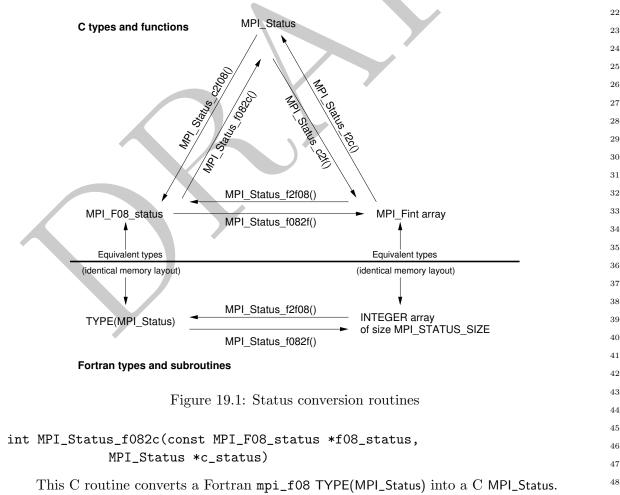
int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)

This call converts a C status into a Fortran status, and has a behavior similar to MPI_Status_f2c . That is, the value of c_status must not be either MPI_STATUS_IGNORE or $MPI_STATUSES_IGNORE$.

Advice to users. There exists no separate conversion function for arrays of statuses, since one can simply loop through the array, converting each status with the routines in Figure 19.1. (*End of advice to users.*)

Rationale. The handling of MPI_STATUS_IGNORE is required in order to layer libraries with only a C wrapper: if the Fortran call has passed MPI_STATUS_IGNORE, then the C wrapper must handle this correctly. Note that this constant need not have the same value in Fortran and C. If MPI_Status_f2c were to handle MPI_STATUS_IGNORE, then the type of its result would have to be MPI_Status**, which was considered an inferior solution. (*End of rationale.*)

Using the mpi_f08 Fortran module, a status is declared as TYPE(MPI_Status). The C type MPI_F08_status can be used to pass a Fortran TYPE(MPI_Status) argument into a C routine. Figure 19.1 illustrates all status conversion routines. Some are only available in C, some in both C and the Fortran mpi and mpi_f08 interfaces (but not in the mpif.h interface).



Unofficial Draft for Comment Only

 $\mathbf{2}$

```
CHAPTER 19. LANGUAGE BINDINGS
1
     int MPI_Status_c2f08(const MPI_Status *c_status,
\mathbf{2}
                    MPI_F08_status *f08_status)
3
          This C routine converts a C MPI_Status into a Fortran mpi_f08 TYPE(MPI_Status). Two
4
     global variables of type MPI_F08_status*, MPI_F08_STATUS_IGNORE and
5
     MPI_F08_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether
6
     f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE defined in
7
     the mpi_f08 module. These are global variables, not C constant expressions and cannot be
8
     used in places where C requires constant expressions. Their value is defined only between
9
     the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code.
10
          Conversion between the two Fortran versions of a status can be done with:
11
12
13
     MPI_STATUS_F2F08(f_status, f08_status)
14
       IN
                f_status
                                            status object declared as array
15
16
       OUT
                 f08_status
                                            status object declared as named type
17
18
     C binding
19
     int MPI_Status_f2f08(const MPI_Fint *f_status, MPI_F08_status *f08_status)
20
     Fortran 2008 binding
21
     MPI_Status_f2f08(f_status, f08_status, ierror)
22
          INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
23
          TYPE(MPI_Status), INTENT(OUT) :: f08_status
24
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     Fortran binding (the following procedure is not available with mpif.h)
27
     MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
28
          INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
29
          TYPE(MPI_Status) :: F08_STATUS
30
         This routine converts a Fortran INTEGER, DIMENSION (MPI_STATUS_SIZE) status array
^{31}
     into a Fortran mpi_f08 TYPE(MPI_Status).
32
33
34
     MPI_STATUS_F082F(f08_status, f_status)
35
       IN
                 f08_status
                                            status object declared as named type
36
37
       OUT
                 f_status
                                            status object declared as array
38
39
     C binding
40
     int MPI_Status_f082f(const MPI_F08_status *f08_status, MPI_Fint *f_status)
41
     Fortran 2008 binding
42
     MPI_Status_f082f(f08_status, f_status, ierror)
43
          TYPE(MPI_Status), INTENT(IN) :: f08_status
44
          INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
45
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     Fortran binding (the following procedure is not available with mpif.h)
48
```

```
MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
   TYPE(MPI_Status) :: F08_STATUS
   INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
```

This routine converts a Fortran mpi_f08 TYPE(MPI_Status) into a Fortran INTEGER, DIMENSION(MPI_STATUS_SIZE) status array.

19.2.6 MPI Opaque Objects

Unless said otherwise, opaque objects are "the same" in all languages: they carry the same information, and have the same meaning in both languages. The mechanism described in the previous section can be used to pass references to MPI objects from language to language. An object created in one language can be accessed, modified or freed in another language.

We examine below in more detail issues that arise for each type of MPI object.

Datatypes

Datatypes encode the same information in all languages. E.g., a datatype accessor like MPI_TYPE_GET_EXTENT will return the same information in all languages. If a datatype defined in one language is used for a communication call in another language, then the message sent will be identical to the message that would be sent from the first language: the same communication buffer is accessed, and the same representation conversion is performed, if needed. All predefined datatypes can be used in datatype constructors in any language. If a datatype is committed, it can be used for communication in any language.

The function MPI_GET_ADDRESS returns the same value in all languages. Note that we do not require that the constant MPI_BOTTOM have the same value in all languages (see Section 19.2.9).

Example 19.12

```
! FORTRAN CODE
REAL :: R(5)
INTEGER :: TYPE, IERR, AOBLEN(1), AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)
! create an absolute datatype for array R
AOBLEN(1) = 5
CALL MPI_GET_ADDRESS(R, AODISP(1), IERR)
AOTYPE(1) = MPI_REAL
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
CALL C_ROUTINE(TYPE)
/* C code */
void C_ROUTINE(MPI_Fint *ftype)
{
   int count = 5;
   int lens[2] = \{1, 1\};
   MPI_Aint displs[2];
```

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

```
1
         MPI_Datatype types[2], newtype;
2
3
         /* create an absolute datatype for buffer that consists
                                                                            */
4
         /* of count, followed by R(5)
                                                                            */
5
6
         MPI_Get_address(&count, &displs[0]);
7
         displs[1] = 0;
8
         types[0] = MPI_INT;
9
         types[1] = MPI_Type_f2c(*ftype);
10
         MPI_Type_create_struct(2, lens, displs, types, &newtype);
11
         MPI_Type_commit(&newtype);
12
13
         MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
14
         /* the message sent contains an int count of 5, followed
                                                                             */
15
         /* by the 5 REAL entries of the Fortran array R.
16
     }
17
           Advice to implementors. The following implementation can be used: MPI addresses,
18
19
           as returned by MPI_GET_ADDRESS, will have the same value in all languages. One
           obvious choice is that MPI addresses be identical to regular addresses. The address
20
           is stored in the datatype, when datatypes with absolute addresses are constructed.
21
           When a send or receive operation is performed, then addresses stored in a datatype
22
           are interpreted as displacements that are all augmented by a base address. This base
23
           address is (the address of) buf, or zero, if buf = MPI_BOTTOM. Thus, if MPI_BOTTOM
24
           is zero then a send or receive call with buf = MPI_BOTTOM is implemented exactly as
25
26
           a call with a regular buffer argument: in both cases the base address is buf. On the
           other hand, if MPL_BOTTOM is not zero, then the implementation has to be slightly
27
           different. A test is performed to check whether buf = MPI_BOTTOM. If true, then the
28
           base address is zero, otherwise it is buf. In particular, if MPI_BOTTOM does not have
29
           the same value in Fortran and C, then an additional test for buf = MPI_BOTTOM is
30
           needed in at least one of the languages.
31
32
           It may be desirable to use a value other than zero for MPI_BOTTOM even in C, so as
33
           to distinguish it from a NULL pointer. If MPI_BOTTOM = c then one can still avoid
34
           the test buf = MPI_BOTTOM, by using the displacement from MPI_BOTTOM, i.e., the
35
           regular address - c, as the MPI address returned by MPI_GET_ADDRESS and stored
36
           in absolute datatypes. (End of advice to implementors.)
37
38
     Callback Functions
39
     MPI calls may associate callback functions with MPI objects: error handlers are associated
40
     with communicators, files, windows, and sessions; attribute copy and delete functions are
41
     associated with attribute keys; reduce operations are associated with operation objects, etc.
42
     In a multilanguage environment, a function passed in an MPI call in one language may be
43
     invoked by an MPI call in another language. MPI implementations must make sure that
44
     such invocation will use the calling convention of the language the function is bound to.
45
46
           Advice to implementors.
                                      Callback functions need to have a language tag. This
47
           tag is set when the callback function is passed in by the library function (which is
48
```

presumably different for each language and language support method), and is used to generate the right calling sequence when the callback function is invoked. (*End of advice to implementors.*)

Advice to users. If a subroutine written in one language or Fortran support method wants to pass a callback routine including the predefined Fortran functions (e.g., MPI_COMM_NULL_COPY_FN) to another application routine written in another language or Fortran support method, then it must be guaranteed that both routines use the callback interface definition that is defined for the argument when passing the callback to an MPI routine (e.g., MPI_COMM_CREATE_KEYVAL); see also the advice to users on page 333. (*End of advice to users.*)

Error Handlers

Advice to implementors. Error handlers, have, in C, a variable length argument list. It might be useful to provide to the handler information on the language environment where the error occurred. (*End of advice to implementors.*)

Reduce Operations

All predefined named and unnamed datatypes as listed in Section 6.9.2 can be used in the listed predefined operations independent of the programming language from which the MPI routine is called.

Advice to users. Reduce operations receive as one of their arguments the datatype of the operands. Thus, one can define "polymorphic" reduce operations that work for C and Fortran datatypes. (*End of advice to users.*)

19.2.7 Attributes

Attribute keys can be allocated in one language and freed in another. Similarly, attribute values can be set in one language and accessed in another. To achieve this, attribute keys will be allocated in an integer range that is valid all languages. The same holds true for system-defined attribute values (such as MPI_TAG_UB, MPI_WTIME_IS_GLOBAL, etc.).

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPI_XXX_CREATE_KEYVAL call). When a communicator is duplicated, for each attribute, the corresponding copy function is called, using the right calling convention for the language of that function; and similarly, for the delete callback function.

Advice to implementors. This requires that attributes be tagged either as "C" or "Fortran" and that the language tag be checked in order to use the right calling convention for the callback function. (*End of advice to implementors.*)

The attribute manipulation functions described in Section 7.7 defines attributes arguments to be of type void* in C, and of type INTEGER, in Fortran. On some systems, INTEGERs will have 32 bits, while C pointers will have 64 bits. This is a problem if communicator attributes are used to move information from a Fortran caller to a C callee, or vice-versa.

Unofficial Draft for Comment Only

 $\overline{7}$

 31

¹ MPI behaves as if it stores, internally, address sized attributes. If Fortran INTEGERs ² are smaller, then the (deprecated) Fortran function MPI_ATTR_GET will return the least ³ significant part of the attribute word; the (deprecated) Fortran function MPI_ATTR_PUT ⁴ will set the least significant part of the attribute word, which will be sign extended to the ⁵ entire word. (These two functions may be invoked explicitly by user code, or implicitly, by ⁶ attribute copying callback functions.)

As for addresses, new functions are provided that manipulate Fortran address sized
 attributes, and have the same functionality as the old functions in C. These functions are
 described in Section 7.7. Users are encouraged to use these new functions.

10 MPI supports two types of attributes: address-valued (pointer) attributes, and integer-11valued attributes. C attribute functions put and get address-valued attributes. Fortran 12attribute functions put and get integer-valued attributes. When an integer-valued attribute 13is accessed from C, then MPI_XXX_get_attr will return the address of (a pointer to) the 14integer-valued attribute, which is a pointer to MPI_Aint if the attribute was stored with 15Fortran MPI_XXX_SET_ATTR, and a pointer to int if it was stored with the deprecated 16Fortran MPI_ATTR_PUT. When an address-valued attribute is accessed from Fortran, then 17MPI_XXX_GET_ATTR will convert the address into an integer and return the result of this 18 conversion. This conversion is lossless if new style attribute functions are used, and an 19integer of kind MPI_ADDRESS_KIND is returned. The conversion may cause truncation if 20deprecated attribute functions are used. In C, the deprecated routines MPI_Attr_put and 21MPI_Attr_get behave identical to MPI_Comm_set_attr and MPI_Comm_get_attr. 22

```
23 Example 19.13
```

 24

34

35 36

37

A. Setting an attribute value in C

```
<sup>25</sup>
<sub>26</sub> int set_val = 3;
```

```
struct foo set_struct;
```

```
_{29}^{28} /* Set a value that is a pointer to an int */
```

```
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval1, &set_val);
/* Set a value that is a pointer to a struct */
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval2, &set_struct);
```

```
/* Set an integer value */
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval3, (void *) 17);
```

```
B. Reading the attribute value in C
```

```
<sup>38</sup> int flag, *get_val;
<sup>39</sup> struct foo *get struct:
```

```
39 struct foo *get_struct;
40
```

```
<sup>41</sup> /* Upon successful return, get_val == &set_val
(and therefore *get_val == 3) */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &get_val, &flag);
/* Upon successful return, get_struct == &set_struct */
MPI_Comm_get_attr(MPI_COMM_WORLD_keyval2_&get_struct_&flag);
```

```
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &get_struct, &flag);
```

```
46 /* Upon successful return, get_val == (void*) 17 */
```

```
<sup>47</sup> /* i.e., (MPI_Aint) get_val == 17 */
<sup>48</sup> MPI Comm get attr(MPI_COMM_WORLD_keyval3_4)
```

```
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval3, &get_val, &flag);
```

```
1
    C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
                                                                                      2
LOGICAL FLAG
INTEGER IERR, GET_VAL, GET_STRUCT
! Upon successful return, GET_VAL == &set_val, possibly truncated
                                                                                      6
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
! Upon successful return, GET_STRUCT == &set_struct, possibly truncated
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
                                                                                      9
! Upon successful return, GET_VAL == 17
                                                                                      10
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
                                                                                      11
                                                                                      12
    D. Reading the attribute value with Fortran MPI-2 calls
                                                                                      13
                                                                                      14
LOGICAL FLAG
                                                                                      15
INTEGER IERR
                                                                                      16
INTEGER (KIND=MPI_ADDRESS_KIND) GET_VAL, GET_STRUCT
                                                                                      17
                                                                                      18
! Upon successful return, GET_VAL == &set_val
                                                                                      19
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
                                                                                      20
! Upon successful return, GET_STRUCT == &set_struct
                                                                                      21
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
                                                                                      22
! Upon successful return, GET_VAL == 17
                                                                                      23
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
                                                                                      ^{24}
                                                                                      25
                                                                                      26
Example 19.14 A. Setting an attribute value with the (deprecated) Fortran MPI-1 call
                                                                                      27
INTEGER IERR, VAL
                                                                                      28
VAL = 7
                                                                                      29
CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEYVAL, VAL, IERR)
                                                                                      30
                                                                                      31
    B. Reading the attribute value in C
                                                                                      32
                                                                                      33
int flag;
                                                                                      34
int *value;
                                                                                      35
                                                                                      36
/* Upon successful return, value points to internal MPI storage and
                                                                                      37
   *value == (int) 7 */
                                                                                      38
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &value, &flag);
                                                                                      39
                                                                                      40
    C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
                                                                                      41
                                                                                      42
LOGICAL FLAG
                                                                                      43
INTEGER IERR, VALUE
                                                                                      44
                                                                                      45
! Upon successful return, VALUE == 7
                                                                                      46
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
                                                                                      47
                                                                                      48
    D. Reading the attribute value with Fortran MPI-2 calls
```

```
1
     LOGICAL FLAG
\mathbf{2}
     INTEGER IERR
3
     INTEGER (KIND=MPI_ADDRESS_KIND) VALUE
4
\mathbf{5}
     ! Upon successful return, VALUE == 7 (sign extended)
6
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
7
8
     Example 19.15 A. Setting an attribute value via a Fortran MPI-2 call
9
10
     INTEGER IERR
     INTEGER(KIND=MPI_ADDRESS_KIND) VALUE1
11
     INTEGER(KIND=MPI_ADDRESS_KIND) VALUE2
12
     VALUE1 = 42
13
     VALUE2 = INT(2, KIND=MPI_ADDRESS_KIND) ** 40
14
15
16
     CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, IERR)
17
     CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, IERR)
18
         B. Reading the attribute value in C
19
20
     int flag;
     MPI_Aint *value1, *value2;
21
22
     /* Upon successful return, value1 points to internal MPI storage and
23
^{24}
        *value1 == 42 */
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &value1, &flag);
25
26
     /* Upon successful return, value2 points to internal MPI storage and
        *value2 == 2^40 */
27
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &value2, &flag);
28
29
         C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
30
^{31}
     LOGICAL FLAG
32
     INTEGER IERR, VALUE1, VALUE2
33
34
     ! Upon successful return, VALUE1 == 42
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
35
36
     ! Upon successful return, VALUE2 == 2<sup>40</sup>, or 0 if truncation
37
     ! needed (i.e., the least significant part of the attribute word)
38
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
39
         D. Reading the attribute value with Fortran MPI-2 calls
40
41
     LOGICAL FLAG
42
     INTEGER IERR
43
     INTEGER (KIND=MPI_ADDRESS_KIND) VALUE1, VALUE2
44
45
     ! Upon successful return, VALUE1 == 42
46
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
47
     ! Upon successful return, VALUE2 == 2^40
48
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
```

The predefined MPI attributes can be integer valued or address-valued. Predefined integer valued attributes, such as MPI_TAG_UB, behave as if they were put by a call to the deprecated Fortran routine MPI_ATTR_PUT, i.e., in Fortran, MPI_COMM_GET_ATTR(MPI_COMM_WORLD, MPI_TAG_UB, val, flag, ierr) will return

in val the upper bound for tag value; in C, MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_TAG_UB, &p, &flag) will return in p a pointer to an int containing the upper bound for tag value.

Address-valued predefined attributes, such as MPI_WIN_BASE behave as if they were put by a C call, i.e., in Fortran, MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, val, flag, ierror) will return in val the base address of the window, converted to an integer. In C, MPI_Win_get_attr(win, MPI_WIN_BASE, &p, &flag) will return in p a pointer to the window base, cast to (void *).

Rationale. The design is consistent with the behavior specified for predefined attributes, and ensures that no information is lost when attributes are passed from language to language. Because the language interoperability for predefined attributes was defined based on MPI_ATTR_PUT, this definition is kept for compatibility reasons although the routine itself is now deprecated. (*End of rationale.*)

Advice to implementors. Implementations should tag attributes either as (1) address attributes, (2) as INTEGER(KIND=MPI_ADDRESS_KIND) attributes or (3) as INTEGER attributes, according to whether they were set in (1) C (with MPI_Attr_put or MPI_XXX_set_attr), (2) in Fortran with MPI_XXX_SET_ATTR or (3) with the deprecated Fortran routine MPI_ATTR_PUT. Thus, the right choice can be made when the attribute is retrieved. (End of advice to implementors.)

19.2.8 Extra-State

Extra-state should not be modified by the copy or delete callback functions. (This is obvious from the C binding, but not obvious from the Fortran binding). However, these functions may update state that is indirectly accessed via extra-state. E.g., in C, extra-state can be a pointer to a data structure that is modified by the copy or callback functions; in Fortran, extra-state can be an index into an entry in a COMMON array that is modified by the copy or callback functions. In a multithreaded environment, users should be aware that distinct threads may invoke the same callback function concurrently: if this function modifies state associated with extra-state, then mutual exclusion code must be used to protect updates and accesses to the shared state.

19.2.9 Constants

MPI constants have the same value in all languages, unless specified otherwise. This does not apply to constant handles (MPI_INT, MPI_COMM_WORLD, MPI_ERRORS_RETURN, MPI_SUM, etc.) These handles need to be converted, as explained in Section 19.2.4. Constants that specify maximum lengths of strings (see Section A.1.1 for a listing) have a value one less in Fortran than C since in C the length includes the null terminating character. Thus, these constants represent the amount of space which must be allocated to hold the largest possible such string, rather than the maximum number of printable characters the string could contain.

$47 \\ 48$

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\mathbf{5}$

 $\mathbf{6}$

 $\overline{7}$

 24

Advice to users. This definition means that it is safe in C to allocate a buffer to receive a string using a declaration like

char name [MPI_MAX_OBJECT_NAME];

(End of advice to users.)

Also constant "addresses," i.e., special values for reference arguments that are not handles, such as MPI_BOTTOM or MPI_STATUS_IGNORE may have different values in different languages.

Rationale. The current MPI standard specifies that MPI_BOTTOM can be used in initialization expressions in C, but not in Fortran. Since Fortran does not normally support call by value, then MPI_BOTTOM in Fortran must be the name of a predefined static variable, e.g., a variable in an MPI declared COMMON block. On the other hand, in C, it is natural to take MPI_BOTTOM = 0 (Caveat: Defining MPI_BOTTOM = 0 implies that NULL pointer cannot be distinguished from MPI_BOTTOM; it may be that MPI_BOTTOM = 1 is better. See the advice to implementors in the Datatypes subsection in Section 19.2.6) Requiring that the Fortran and C values be the same will complicate the initialization process. (End of rationale.)

²² 19.2.10 Interlanguage Communication

The type matching rules for communication in MPI are not changed: the datatype specification for each item sent should match, in type signature, the datatype specification used to receive this item (unless one of the types is MPI_PACKED). Also, the type of a message item should match the type declaration for the corresponding communication buffer location, unless the type is MPI_BYTE or MPI_PACKED. Interlanguage communication is allowed if it complies with these rules.

Example 19.16 In the example below, a Fortran array is sent from Fortran and received in C.

```
33
     ! FORTRAN CODE
34
     SUBROUTINE MYEXAMPLE()
35
     USE mpi_f08
36
     REAL :: R(5)
37
     INTEGER :: IERR, MYRANK, AOBLEN(1)
38
     TYPE(MPI_Datatype) :: TYPE, AOTYPE(1)
39
     INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)
40
41
     ! create an absolute datatype for array R
42
     AOBLEN(1) = 5
43
     CALL MPI_GET_ADDRESS(R, AODISP(1), IERR)
^{44}
     AOTYPE(1) = MPI_REAL
45
     CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
46
     CALL MPI_TYPE_COMMIT(TYPE, IERR)
47
48
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
```

1

 $\mathbf{2}$

3 4

5 6

7

8

9

10 11

12

13

14

15

16

17

18

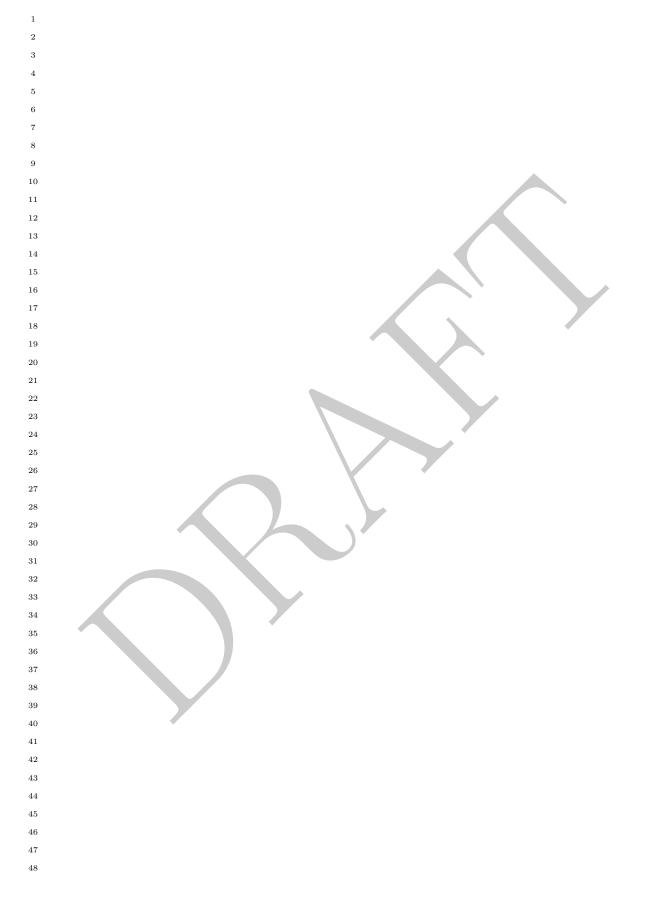
19

```
IF (MYRANK.EQ.0) THEN
CALL MPI_SEND(MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
ELSE
CALL C_ROUTINE(TYPE%MPI_VAL)
END IF
END SUBROUTINE
/* C code */
void C_ROUTINE(MPI_Fint *fhandle)
{
    MPI_Datatype type;
    MPI_Status status;
    type = MPI_Type_f2c(*fhandle);
    MPI_Recv(MPI_BOTTOM, 1, type, 0, 0, MPI_COMM_WORLD, &status);
}
```

MPI implementors may weaken these type matching rules, and allow messages to be sent with Fortran types and received with C types, and vice versa, when those types match. I.e., if the Fortran type INTEGER is identical to the C type int, then an MPI implementation may allow data to be sent with datatype MPI_INTEGER and be received with datatype MPI_INT. However, such code is not portable.

 $\mathbf{2}$

 24



Annex A

Language Bindings Summary

In this section we summarize the specific bindings for C and Fortran. First we present the constants, type definitions, info values and keys. Then we present the routine prototypes separately for each binding. Listings are alphabetical within chapter.

A.1 Defined Values and Handles

A.1.1 Defined Constants

The C and Fortran names are listed below. Constants with the type const int may also be implemented as literal integer constants substituted by the preprocessor.

I		24
	Error classes	25
	C type: const int (or unnamed enum)	26
	Fortran type: INTEGER	27
	MPI_SUCCESS	28
	MPI_ERR_BUFFER	29
	MPI_ERR_COUNT	30
	MPI_ERR_TYPE	31
	MPI_ERR_TAG	32
	MPI_ERR_COMM	33
	MPI_ERR_RANK	34
	MPI_ERR_REQUEST	35
	MPI_ERR_ROOT	36
	MPI_ERR_GROUP	37
	MPI_ERR_OP	38
	MPI_ERR_TOPOLOGY	39
	MPI_ERR_DIMS	40
	MPI_ERR_ARG	41
	MPI_ERR_UNKNOWN	42
	MPI_ERR_TRUNCATE	43
	MPI_ERR_OTHER	44
	MPI_ERR_INTERN	45
	MPI_ERR_PENDING	46
	(Continued on next page)	47
		48

1	Error classes (continued)
2	C type: const int (or unnamed enum)
3	Fortran type: INTEGER
4	MPI_ERR_IN_STATUS
5	MPI_ERR_ACCESS
6	MPI_ERR_AMODE
7	MPI_ERR_ASSERT
8	MPI_ERR_BAD_FILE
9	MPI_ERR_BASE
10	MPI_ERR_CONVERSION
11	MPI_ERR_DISP
12	MPI_ERR_DUP_DATAREP
13	MPI_ERR_FILE_EXISTS
14	MPI_ERR_FILE_IN_USE
15	MPI_ERR_FILE
16	MPI_ERR_INFO_KEY
17	MPI_ERR_INFO_NOKEY
18	MPI_ERR_INFO_VALUE
19	MPI_ERR_INFO
20	MPI_ERR_IO
21	MPI_ERR_KEYVAL
22	MPI_ERR_LOCKTYPE
23	MPI_ERR_NAME
24	MPI_ERR_NO_MEM
25	MPI_ERR_NOT_SAME
26	MPI_ERR_NO_SPACE
27	MPI_ERR_NO_SUCH_FILE
28	MPI_ERR_PORT
29	MPI_ERR_PROC_ABORTED
30	MPI_ERR_QUOTA
31	MPI_ERR_READ_ONLY
32	MPI_ERR_RMA_ATTACH
33	MPI_ERR_RMA_CONFLICT
34	MPI_ERR_RMA_RANGE
35	MPI_ERR_RMA_SHARED
36	MPI_ERR_RMA_SYNC
37	MPI_ERR_RMA_FLAVOR
38	MPI_ERR_SERVICE
39	MPI_ERR_SIZE
40	MPI_ERR_SPAWN
41	MPI_ERR_UNSUPPORTED_DATAREP
42	MPI_ERR_UNSUPPORTED_OPERATION
43	MPI_ERR_WIN
44	(Continued on next page)
45	(Continued on next page)
46	
47	
48	

	Error classes (continued)	1
-	C type: const int (or unnamed enum)	2
	Fortran type: INTEGER	3
-	MPI_T_ERR_CANNOT_INIT	4
	MPI_T_ERR_NOT_ACCESSIBLE	5
	MPI_T_ERR_NOT_INITIALIZED	6
	MPI_T_ERR_NOT_SUPPORTED	7
	MPI_T_ERR_MEMORY	8
	MPI_T_ERR_INVALID	9
	MPI_T_ERR_INVALID_INDEX	10
	MPI_T_ERR_INVALID_ITEM	11
	MPI_T_ERR_INVALID_SESSION	12
	MPI_T_ERR_INVALID_HANDLE	13
	MPI_T_ERR_INVALID_NAME	14
	MPI_T_ERR_OUT_OF_HANDLES	15
	MPI_T_ERR_OUT_OF_SESSIONS	16
	MPI_T_ERR_CVAR_SET_NOT_NOW	17
	MPI_T_ERR_CVAR_SET_NEVER	18
	MPI_T_ERR_PVAR_NO_WRITE	19
	MPI_T_ERR_PVAR_NO_STARTSTOP	20
	MPI_T_ERR_PVAR_NO_ATOMIC	21
	MPI_ERR_LASTCODE	22
-		23
	Buffer Address Constants	24
C type: void * c	const	25
Fortran type: (pre	edefined memory location) ¹	26
MPI_BOTTOM		27
MPI_IN_PLACE		28
	ortran these constants are not usable for initialization	29
expressions or	assignment. See Section 2.5.4.	30
		31
-	Assorted Constants	32
	C type: const int (or unnamed enum)	33
	Fortran type: INTEGER	34
	MPI_PROC_NULL	35
	MPI_ANY_SOURCE	36
	MPI_ANY_TAG	37 38
		39
	MPI_BSEND_OVERHEAD	40
		40
		42
	MPI_LOCK_SHARED	43
	MPI_ROOT	43
	No Process Mossage Handle	45
	No Process Message Handle	46
	ype: MPI_Message rtran type: INTEGER or TYPE(MPI_Message)	47
	PI_MESSAGE_NO_PROC	48

1	Fortran Support Method Specific Constants
2	Fortran type: LOGICAL
3	MPI_SUBARRAYS_SUPPORTED (Fortran only)
4	MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)
5	MPLASTIC_PROTECTS_NONBLOCKING (FORTALI OILY)
6	Status array size and reserved index values (Fortran only)
7	Fortran type: INTEGER
8	MPI_STATUS_SIZE
9	MPI_SOURCE
10	MPI_TAG
11	MPI_ERROR
12	
13	
14	Fortran status array size and reserved index values (C only)
15	C type: int
16	MPI_F_STATUS_SIZE
17	MPI_F_SOURCE
18	MPI_F_TAG
19	MPI_F_ERROR
20	
21	Variable Address Size (Fortran only)
22	Fortran type: INTEGER
23	MPI_ADDRESS_KIND
24	MPI_COUNT_KIND
25	MPI_INTEGER_KIND
26	MPI_OFFSET_KIND
27	
28	Error-handling specifiers
29	C type: MPI_Errhandler
30	Fortran type: INTEGER or TYPE(MPI_Errhandler)
31	MPI_ERRORS_ARE_FATAL
32	MPI_ERRORS_RETURN
33	
34	Maximum Sizes for Strings
35	C type: const int (or unnamed enum)
36	Fortran type: INTEGER
37	MPI_MAX_DATAREP_STRING
38	MPI_MAX_ERROR_STRING
39	MPI_MAX_INFO_KEY
40	MPI_MAX_INFO_VAL
41	MPI_MAX_LIBRARY_VERSION_STRING
42	MPI_MAX_OBJECT_NAME
43	MPI_MAX_PORT_NAME
44	MPI_MAX_PROCESSOR_NAME
45	MPI_MAX_FROM_GROUP_TAG
46	MPI_MAX_PSET_NAME_LEN
47	
48	

Named Predefined Datatypes	C types
$\mathrm{C}\ \mathrm{type}$: MPI_Datatype	
Fortran type: INTEGER	
or TYPE(MPI_Datatype)	
MPI_CHAR	char
	(treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long
MPI_LONG_LONG_INT	signed long long
$MPI_LONG_LONG \ (\mathrm{as} \ \mathrm{a} \ \mathrm{synonym})$	signed long long
MPI_SIGNED_CHAR	signed char
	(treated as integral value)
MPI_UNSIGNED_CHAR	unsigned char
	(treated as integral value)
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_LONG_LONG	unsigned long long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t
-	(defined in <stddef.h>)</stddef.h>
	(treated as printable character)
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_AINT	MPI_Aint
MPI_COUNT	MPI_Count
MPI_OFFSET	MPI_Offset
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	(any C type)

1	Named Predefined Datatypes	Fortran types
2	C type: MPI_Datatype	
3	Fortran type: INTEGER	
4	or TYPE(MPI_Datatype)	
5	MPI_INTEGER	INTEGER
6	MPI_REAL	REAL
7	MPI_DOUBLE_PRECISION	DOUBLE PRECISION
8	MPI_COMPLEX	COMPLEX
9	MPI_LOGICAL	LOGICAL
10	MPI_CHARACTER	CHARACTER(1)
11	MPI_AINT	INTEGER (KIND=MPI_ADDRESS_KIND)
12	_ MPI_COUNT	INTEGER (KIND=MPI_COUNT_KIND)
13	MPI_OFFSET	INTEGER (KIND=MPI_OFFSET_KIND)
14	MPI_BYTE	(any Fortran type)
15	_ MPI_PACKED	(any Fortran type)
16		
17	Named Predefined Datatype	$s^1 \mid C++ types$
18	C type: MPI_Datatype	
19	Fortran type: INTEGER	
20	or TYPE(MPI_Datatype)	
21	MPI_CXX_BOOL	bool
22	MPI_CXX_FLOAT_COMPLEX	<pre>std::complex<float></float></pre>
23	MPI_CXX_DOUBLE_COMPLEX	<pre>std::complex<double></double></pre>
24	MPI_CXX_LONG_DOUBLE_COMPL	
25	^{1} If an accompanying C++ comp	oiler is missing, then the
26	MPI datatypes in this table are	not defined.
27		
28	Optional datatypes (F	ortran) Fortran types
29	C type: MPI_Datatype	
30	Fortran type: INTEGER	
31	or TYPE(MPI_Datatype)	
32	MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
33	MPI_INTEGER1	INTEGER*1
34	MPI_INTEGER2	INTEGER*2
35	MPI_INTEGER4	INTEGER*4
36	MPI_INTEGER8	INTEGER*8
37	MPI_INTEGER16	INTEGER*16
38	MPI_REAL2	REAL*2
39	MPI_REAL4	REAL*4
40	MPI_REAL8	REAL*8
41	MPI_REAL16	REAL*16
42	MPI_COMPLEX4	COMPLEX*4
43	MPI_COMPLEX8	COMPLEX*8
44	MPI_COMPLEX16	COMPLEX*16
45	MPI_COMPLEX32	COMPLEX*32
46		
47		
48		

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
4 5 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 11 12 13 14 15 16 17 18 19 20 21 22 23
12 13 14 15 16 17 18 19 20 21 22 23
 13 14 15 16 17 18 19 20 21 22 23
14 15 16 17 18 19 20 21 22 23
15 16 17 18 19 20 21 22 23
 16 17 18 19 20 21 22 23
 17 18 19 20 21 22 23
18 19 20 21 22 23
19 20 21 22 23
20 21 22 23
21 22 23
22 23
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
A 14
45
45 46

1	Collective Operations
2	C type: MPI_Op
3	Fortran type: INTEGER or TYPE(MPI_Op)
4	MPI_MAX
5	MPI_MIN
6	MPI_SUM
7	MPI_PROD
8	MPI_MAXLOC
9	MPI_MINLOC
10	MPI_BAND
11	MPI_BOR
12	MPI_BXOR
13	MPI_LAND
14	 MPI_LOR
15	MPI_LXOR
16	MPI_REPLACE
17	MPI_NO_OP
18	
19	Null Handles
20	C/Fortran name
21	C type / Fortran type
22	MPI_GROUP_NULL
23	MPI_Group / INTEGER or TYPE(MPI_Group)
24	MPI_COMM_NULL
25	MPI_Comm / INTEGER or TYPE(MPI_Comm)
26	MPI_DATATYPE_NULL
27	MPI_Datatype / INTEGER or TYPE(MPI_Datatype)
28	MPI_REQUEST_NULL
29	MPI_Request / INTEGER or TYPE(MPI_Request)
30	MPI_OP_NULL
31	MPI_Op / INTEGER or TYPE(MPI_Op)
32	MPI_ERRHANDLER_NULL
33	MPI_Errhandler / INTEGER or TYPE(MPI_Errhandler)
34	MPI_FILE_NULL
35	MPI_File / INTEGER or TYPE(MPI_File)
36	MPI_INFO_NULL
37	MPI_Info / INTEGER or TYPE(MPI_Info)
38	MPI_SESSION_NULL
39	MPI_Session / INTEGER
40	MPI_WIN_NULL
41	MPI_WIN_INTEGER or TYPE(MPI_Win)
42	MPI_MESSAGE_NULL
43	MPI_MESSAGE_NOLL MPI_Message / INTEGER or TYPE(MPI_Message)
44	
45	Empty group
46	C type: MPI_Group
47	Fortran type: INTEGER or TYPE(MPI_Group)
48	MPI_GROUP_EMPTY

	Topologies
	C type: const int (or unnamed enum)
	Fortran type: INTEGER
	MPI_GRAPH
	MPI_CART
	MPI_DIST_GRAPH
	Predefined functions
C/Fortran name	Predefined functions
C type	
/ Fortran type with mp	i module / Fortran type with mpi_f08 module
MPI_COMM_NULL_CO	
MPI_Comm_copy_attr_:	
/ COMM_COPY_ATTR_FUN	
MPI_COMM_DUP_FN	
MPI_Comm_copy_attr_	function
/ COMM_COPY_ATTR_FUN	ICTION / PROCEDURE(MPI_Comm_copy_attr_function) 1)
MPI_COMM_NULL_DEI	LETE_FN
MPI_Comm_delete_att:	r_function
/ COMM_DELETE_ATTR_F	, , , , , , , , , , , , , , , , , , , ,
MPI_WIN_NULL_COPY	_FN
MPI_Win_copy_attr_f	
/ WIN_COPY_ATTR_FUNC	$TION / PROCEDURE(MPI_Win_copy_attr_function)$
MPI_WIN_DUP_FN	
MPI_Win_copy_attr_f	
/ WIN_COPY_ATTR_FUNC	
MPI_WIN_NULL_DELE	
MPI_Win_delete_attr	
/ WIN_DELETE_ATTR_FU	
MPI_TYPE_NULL_COP	
MPI_Type_copy_attr_: / TYPE_COPY_ATTR_FUN	
MPI_TYPE_DUP_FN	<pre>ICTION / PROCEDURE(MPI_Type_copy_attr_function) 1)</pre>
MPI_Type_copy_attr_:	function
/ TYPE_COPY_ATTR_FUN	
MPI_TYPE_NULL_DELI	
MPI_Type_delete_att	
/ TYPE_DELETE_ATTR_F	
MPI_CONVERSION_FN	
MPI_Datarep_convers	
-	
/ DATAREP_CONVERSION	ementors (on page 333) and advice to users (on page 333)
,	ementors (on page 333) and advice to users (on page 333)
^{1} See the advice to impl	rtran functions MPI_COMM_NULL_COPY_FN, in

1	Deprecated predefined functions
2	C/Fortran name
3	C type / Fortran type with mpi module
4	MPI_NULL_COPY_FN
5	MPI_Copy_function / COPY_FUNCTION
6	MPI_DUP_FN
7	MPI_Copy_function / COPY_FUNCTION
8	MPI_NULL_DELETE_FN
9	MPI_Delete_function / DELETE_FUNCTION
10	
11	Predefined Attribute Keys
12	C type: const int (or unnamed enum)
13	Fortran type: INTEGER
14	MPI_APPNUM
15	MPI_LASTUSEDCODE
16	MPI_UNIVERSE_SIZE
17	MPI_WIN_BASE
18	MPI_WIN_DISP_UNIT
19	MPI_WIN_SIZE
20	MPI_WIN_CREATE_FLAVOR
21	MPI_WIN_MODEL
22	
23	MPI Window Create Flavors
24	C type: const int (or unnamed enum)
25	Fortran type: INTEGER
26	MPI_WIN_FLAVOR_CREATE
27	MPI_WIN_FLAVOR_ALLOCATE
28	MPI_WIN_FLAVOR_DYNAMIC
29	MPI_WIN_FLAVOR_SHARED
30	
31	MPI Window Models
32	C type: const int (or unnamed enum)
33	Fortran type: INTEGER
34	MPI_WIN_SEPARATE
35	MPI_WIN_UNIFIED
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

	Mode Constants
-	C type: const int (or unnamed enum)
	Fortran type: INTEGER
_	MPI_MODE_APPEND
	MPI_MODE_CREATE
	MPI_MODE_DELETE_ON_CLOSE
	MPI_MODE_EXCL
	MPI_MODE_NOCHECK
	MPI_MODE_NOPRECEDE
	MPI_MODE_NOPUT
	MPI_MODE_NOSTORE
	MPI_MODE_NOSUCCEED
	MPI_MODE_RDONLY
	MPI_MODE_RDWR
	MPI_MODE_SEQUENTIAL
	MPI_MODE_UNIQUE_OPEN
	MPI_MODE_WRONLY
_	Datatype Decoding Constants
	C type: const int (or unnamed enum)
_	Fortran type: INTEGER
	MPI_COMBINER_CONTIGUOUS
	MPI_COMBINER_DARRAY
	MPI_COMBINER_DUP
	MPI_COMBINER_F90_COMPLEX
	MPI_COMBINER_F90_INTEGER
	MPI_COMBINER_F90_REAL
	MPI_COMBINER_HVECTOR
	MPI_COMBINER_INDEXED_BLOCK
	MPI_COMBINER_HINDEXED_BLOCK
	MPI_COMBINER_INDEXED
	MPI_COMBINER_NAMED MPI_COMBINER_RESIZED
	MPI_COMBINER_STRUCT
	MPI_COMBINER_SUBARRAY
	MPI_COMBINER_VECTOR
-	
	Threads Constants
-	C type: const int (or unnamed enum)
	Fortran type: INTEGER
_	MPI_THREAD_FUNNELED
	MPI_THREAD_MULTIPLE
	MPI_THREAD_SERIALIZED
	MPI_THREAD_SINGLE

	File Operation Constants, Part 1
2	C type: const MPI_Offset (or unnamed enum)
3	Fortran type: INTEGER (KIND=MPI_OFFSET_KIND)
4	MPI_DISPLACEMENT_CURRENT
5	
6	File Operation Constants, Part 2
7	C type: const int (or unnamed enum)
8	Fortran type: INTEGER
9	MPI_DISTRIBUTE_BLOCK
10	MPI_DISTRIBUTE_CYCLIC
11	MPI_DISTRIBUTE_DFLT_DARG
12	MPI_DISTRIBUTE_NONE
13	MPI_ORDER_C
14	MPI_ORDER_FORTRAN
15	MPI_SEEK_CUR
16	MPI_SEEK_END
17	MPI_SEEK_SET
18	
19	F90 Datatype Matching Constants
20	C type: const int (or unnamed enum)
21	Fortran type: INTEGER
22	MPI_TYPECLASS_COMPLEX
23	MPI_TYPECLASS_INTEGER
24	MPI_TYPECLASS_REAL
25	
26	Constants Specifying Empty or Ignored Input
27	C/Fortran name
27 28	C/Fortran name C type / Fortran type ¹
27 28 29	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL
27 28 29 30	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*)
27 28 29 30 31	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim, array of CHARACTER*(*) MPI_ARGV_NULL
27 28 29 30 31 32	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*)
27 28 29 30 31 32 33	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE
27 28 29 30 31 32 33 34	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array
27 28 29 30 31 32 33	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE
27 28 29 30 31 32 33 34 35	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)
27 28 29 30 31 32 33 34 35 36	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*)
27 28 29 30 31 32 33 34 35 36 37	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE
27 28 29 30 31 32 33 34 35 36 37 38	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)
27 28 29 30 31 32 33 34 35 36 37 38 39	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status)
27 28 29 30 31 32 33 34 35 36 37 38 39 40	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED int* / INTEGER array
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED int* / INTEGER array MPI_WEIGHTS_EMPTY
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED int* / INTEGER array MPI_WEIGHTS_EMPTY int* / INTEGER array
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim, array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED int* / INTEGER array MPI_WEIGHTS_EMPTY int* / INTEGER array ¹ Note that in Fortran these constants are not usable for initialization
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	C/Fortran name C type / Fortran type ¹ MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED int* / INTEGER array MPI_WEIGHTS_EMPTY int* / INTEGER array

I_F_STATUS_IGNOREMPI_STATUS_IGNORE in mpi / mpif.hype: MPI_F08_status*equivalent to FortranI_F08_STATUSES_IGNOREMPI_STATUSES_IGNORE in mpi_f08		
ype: MPI_F08_status* equivalent to Fortran I_F08_STATUSES_IGNORE MPI_STATUSES_IGNORE in mpi_f08 I_F08_STATUS_IGNORE MPI_STATUS_IGNORE in mpi_f08 C preprocessor Constants and Fortran Parameters C type: C-preprocessor macro that expands to an int value Fortran type: INTEGER MPI_SUBVERSION MPI_VERSION MPI_VERSION MPI_T_ENUM_NULL MPI_T_enum MPI_T_CVAR_HANDLE_NULL MPI_T_CVAR_HANDLE_NULL MPI_T_PVAR_HANDLE_NULL MPI_T_PVAR_SESSION_NULL MPI_T_PVAR_SESSION_NULL MPI_T_PVAR_SESSION_NULL MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL MPI_T_VERBOSITY_MPIDEV_DETAIL		<code>PI_STATUSES_IGNORE</code> in <code>mpi</code> / <code>mpif.h</code>
I_F08_STATUSES_IGNORE MPI_STATUSES_IGNORE in mpi_f08 I_F08_STATUS_IGNORE MPI_STATUS_IGNORE in mpi_f08 I_F08_STATUS_IGNORE MPI_STATUS_IGNORE in mpi_f08 C preprocessor Constants and Fortran Parameters C type: C-preprocessor macro that expands to an int value Fortran type: INTEGER MPI_SUBVERSION MPI_VERSION MPI_VERSION MPI_VERSION MPI_T_ENUM_NULL MPI_T_enum MPI_T_CVAR_HANDLE_NULL MPI_T_ovar_handle MPI_T_PVAR_HANDLE_NULL MPI_T_pvar_handle MPI_T_PVAR_SESSION_NULL MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const_int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL	PI_F_STATUS_IGNORE MI	<pre>PI_STATUS_IGNORE in mpi / mpif.h</pre>
I_F08_STATUS_IGNORE MPI_STATUS_IGNORE in mpi_f08 C preprocessor Constants and Fortran Parameters C type: C-preprocessor macro that expands to an int value Fortran type: INTEGER MPI_SUBVERSION MPI_VERSION MPI_VERSION MPI_T_ENUM_NULL MPI_T_enum MPI_T_CVAR_HANDLE_NULL MPI_T_PVAR_HANDLE_NULL MPI_T_pvar_handle MPI_T_PVAR_SESSION_NULL MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const_int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL	eq eq	uivalent to Fortran
C preprocessor Constants and Fortran Parameters C type: C-preprocessor macro that expands to an int value Fortran type: INTEGER MPI_SUBVERSION MPI_VERSION MPI_VERSION MPI_T_ENUM_NULL MPI_T_enum MPI_T_CVAR_HANDLE_NULL MPI_T_cvar_handle MPI_T_PVAR_HANDLE_NULL MPI_T_PVAR_SESSION_NULL MPI_T_PVAR_SESSION_NULL MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL MPI_T_VERBOSITY_MPIDEV_DETAIL	PI_F08_STATUSES_IGNORE MI	<code>Pl_STATUSES_IGNORE</code> in <code>mpi_f08</code>
C type: C-preprocessor macro that expands to an int value Fortran type: INTEGER MPI_SUBVERSION MPI_VERSION MPI_VERSION MPI_T_ENUM_NULL MPI_T_enum MPI_T_CVAR_HANDLE_NULL MPI_T_PVAR_HANDLE_NULL MPI_T_Pvar_handle MPI_T_Pvar_session Verbosity Levels in the MPI tool information interface C type: const_int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL	PI_F08_STATUS_IGNORE MI	PI_STATUS_IGNORE in mpi_f08
C type: C-preprocessor macro that expands to an int value Fortran type: INTEGER MPI_SUBVERSION MPI_VERSION MPI_VERSION MPI_T_ENUM_NULL MPI_T_enum MPI_T_CVAR_HANDLE_NULL MPI_T_PVAR_HANDLE_NULL MPI_T_Pvar_handle MPI_T_Pvar_session Verbosity Levels in the MPI tool information interface C type: const_int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
Fortran type: INTEGER MPI_SUBVERSION MPI_VERSION MPI_VERSION Null handles used in the MPI tool information interface MPI_T_ENUM_NULL MPI_T_enum MPI_T_CVAR_HANDLE_NULL MPI_T_CVAR_HANDLE_NULL MPI_T_PVAR_HANDLE_NULL MPI_T_PVAR_SESSION_NULL MPI_T_PVAR_SESSION_NULL MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const_int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_DETAIL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_DETAIL MPI_T_VERBOSITY_MPIDEV_DETAIL		
MPI_SUBVERSION MPI_VERSION MPI_VERSION MPI_VERSION MPI_T_ENUM_NULL MPI_T_Enum MPI_T_CVAR_HANDLE_NULL MPI_T_pvar_handle MPI_T_PVAR_HANDLE_NULL MPI_T_pvar_handle MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		o that expands to an int value
MPI_VERSION Null handles used in the MPI tool information interface MPI_T_ENUM_NULL MPI_T_Enum MPI_T_CVAR_HANDLE_NULL MPI_T_CVAR_HANDLE_NULL MPI_T_PVAR_HANDLE_NULL MPI_T_PVAR_SESSION_NULL MPI_T_PVAR_SESSION_NULL MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
Null handles used in the MPI tool information interface MPI_T_ENUM_NULL MPI_T_enum MPI_T_CVAR_HANDLE_NULL MPI_T_PVAR_HANDLE_NULL MPI_T_pvar_handle MPI_T_PVAR_SESSION_NULL MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const_int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
MPI_T_ENUM_NULL MPI_T_enum MPI_T_CVAR_HANDLE_NULL MPI_T_cvar_handle MPI_T_PVAR_HANDLE_NULL MPI_T_pvar_handle MPI_T_PVAR_SESSION_NULL MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_DETAIL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL	MPI_VERSION	
MPI_T_ENUM_NULL MPI_T_enum MPI_T_CVAR_HANDLE_NULL MPI_T_cvar_handle MPI_T_PVAR_HANDLE_NULL MPI_T_pvar_handle MPI_T_PVAR_SESSION_NULL MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_DETAIL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL	Null handles used in the	MPI tool information interface
MPI_T_enum MPI_T_CVAR_HANDLE_NULL MPI_T_cvar_handle MPI_T_PVAR_HANDLE_NULL MPI_T_pvar_handle MPI_T_PVAR_SESSION_NULL MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const_int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_DETAIL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
MPI_T_CVAR_HANDLE_NULL MPI_T_cvar_handle MPI_T_PVAR_HANDLE_NULL MPI_T_pvar_handle MPI_T_PVAR_SESSION_NULL MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const_int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_DETAIL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		Ť
MPI_T_cvar_handle MPI_T_PVAR_HANDLE_NULL MPI_T_pvar_handle MPI_T_PVAR_SESSION_NULL MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const_int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_DETAIL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
MPI_T_PVAR_HANDLE_NULL MPI_T_pvar_handle MPI_T_PVAR_SESSION_NULL MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const_int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_DETAIL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
MPI_T_pvar_handle MPI_T_PVAR_SESSION_NULL MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_DETAIL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
MPI_T_PVAR_SESSION_NULL MPI_T_pvar_session Verbosity Levels in the MPI tool information interface C type: const_int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_DETAIL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
Verbosity Levels in the MPI tool information interface C type: const int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_DETAIL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL	·	
C type: const int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_DETAIL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL	MPI_T_pvar_session	
C type: const int (or unnamed enum) MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_DETAIL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
MPI_T_VERBOSITY_USER_BASIC MPI_T_VERBOSITY_USER_DETAIL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
MPI_T_VERBOSITY_USER_DETAIL MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		· · · · · · · · · · · · · · · · · · ·
MPI_T_VERBOSITY_USER_ALL MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
MPI_T_VERBOSITY_TUNER_BASIC MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
MPI_T_VERBOSITY_TUNER_DETAIL MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
MPI_T_VERBOSITY_TUNER_ALL MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
MPI_T_VERBOSITY_MPIDEV_BASIC MPI_T_VERBOSITY_MPIDEV_DETAIL		
MPI_T_VERBOSITY_MPIDEV_DETAIL		
MPI_I_VERBOSITY_MPIDEV_ALL		-
	MPI_T_VERBOSITY_MPIDEV_	ALL

	808	ANNEX A. LANGUAGE BINDINGS SUMMARY
1		Constants to identify associations of variables
2		in the MPI tool information interface
3		C type: const int (or unnamed enum)
4		MPI_T_BIND_NO_OBJECT
5		MPI_T_BIND_MPI_COMM
6		MPI_T_BIND_MPI_DATATYPE
7		MPI_T_BIND_MPI_ERRHANDLER
8		MPI_T_BIND_MPI_FILE
9		MPI_T_BIND_MPI_GROUP
10		MPI_T_BIND_MPI_OP
11		MPI_T_BIND_MPI_REQUEST
12		MPI_T_BIND_MPI_WIN
13		MPI_T_BIND_MPI_MESSAGE
14		MPI_T_BIND_MPI_INFO
15		
16		Constants describing the scope of a control variable
17		in the MPI tool information interface
18		C type: const int (or unnamed enum)
19		MPI_T_SCOPE_CONSTANT
20 21		MPI_T_SCOPE_READONLY
21		MPI_T_SCOPE_LOCAL
22		MPI_T_SCOPE_GROUP
23		MPI_T_SCOPE_GROUP_EQ
25		MPI_T_SCOPE_ALL
26		MPI_T_SCOPE_ALL_EQ
27		Additional constants used
28		by the MPI tool information interface
29		C type: MPI_T_pvar_handle
30		MPI_T_PVAR_ALL_HANDLES
31		
32		Performance variables classes used by the
33		MPI tool information interface
34		C type: const int (or unnamed enum)
35		MPI_T_PVAR_CLASS_STATE
36		MPI_T_PVAR_CLASS_LEVEL
37		MPI_T_PVAR_CLASS_SIZE
38		MPI_T_PVAR_CLASS_PERCENTAGE
39		MPI_T_PVAR_CLASS_HIGHWATERMARK
40		MPI_T_PVAR_CLASS_LOWWATERMARK
41		MPI_T_PVAR_CLASS_COUNTER
42		MPI_T_PVAR_CLASS_AGGREGATE
43		MPI_T_PVAR_CLASS_TIMER
44		MPI_T_PVAR_CLASS_GENERIC
45		
46		
47		
48		

	Source event ordering guarantees in the	1
	MPI tool information interface	2
	C type: MPI_T_source_order	3
	MPI_T_SOURCE_ORDERED	4
	MPI_T_SOURCE_UNORDERED	5
		6
	Callback safety requirement levels used in the	7
	MPI tool information interface	8
-	C type: MPI_T_cb_safety	9
-	MPI_T_CB_REQUIRE_NONE	10
	MPI_T_CB_REQUIRE_MPI_RESTRICTED	11
	MPI_T_CB_REQUIRE_THREAD_SAFE	12
	MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE	13
-		14
A.1.2 Types		15
		16
The following are	defined C type definitions, included in the file mpi.h.	17
/* C opaque type	es */	18
MPI_Aint		19
MPI_Count		20
MPI_Fint		21
MPI_Offset		22
MPI_Status		23
MPI_F08_status		24
		25
/* C handles to	assorted structures */	26
MPI_Comm		27
MPI_Datatype		28
MPI_Errhandler		29
MPI_File		30
MPI_Group		31
MPI_Info		32
MPI_Message		33
MPI_Op		34
MPI_Request		35
MPI_Session		36
MPI_Win		37
		38
/* Types for th	e MPI_T interface */	39
MPI_T_enum		40
MPI_T_cvar_hand	le	41
MPI_T_pvar_hand	le	42
MPI_T_pvar_sess		43
MPI_T_event_ins	tance	44
MPI_T_event_reg	istration	45
MPI_T_source_or	der	46
MPI_T_cb_safety		47
		48

1 2 3 The following are defined Fortran type definitions, included in the mpi_f08 and mpi 4 modules. 5! Fortran opaque types in the mpi_f08 and mpi modules 6 TYPE(MPI_Status) 7 8 ! Fortran handles in the mpi_f08 and mpi modules 9 TYPE(MPI_Comm) 10 TYPE(MPI_Datatype) 11 TYPE(MPI_Errhandler) 12TYPE(MPI_File) 13 TYPE(MPI_Group) 14 TYPE(MPI_Info) 15 TYPE(MPI_Message) 16 TYPE(MPI_Op) 17TYPE(MPI_Request) 18 TYPE(MPI_Session) 19 TYPE(MPI_Win) 2021A.1.3 Prototype Definitions 22 23C Bindings 24The following are defined C typedefs for user-defined functions, also included in the file 25mpi.h. 2627/* prototypes for user-defined functions */ 28typedef void MPI_User_function(void *invec, void *inoutvec, int *len, 29 MPI_Datatype *datatype); 30 31 typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval, 32 void *extra_state, void *attribute_val_in, 33 void *attribute_val_out, int *flag); 34 typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval, 35void *attribute_val, void *extra_state); 36 37 typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval, 38 void *extra_state, void *attribute_val_in, 39 void *attribute_val_out, int *flag); 40 41 typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval, 42void *attribute_val, void *extra_state); 43 typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype, 44 int type_keyval, void *extra_state, void *attribute_val_in, 45 void *attribute_val_out, int *flag); 46 47typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype, 48 int type_keyval, void *attribute_val, void *extra_state);

typedef	<pre>void MPI_Comm_errhandler_function(MPI_Comm *comm, int *error_code,</pre>	1 2
typedef	<pre>void MPI_Win_errhandler_function(MPI_Win *win, int *error_code,);</pre>	3 4 5
typedef	<pre>void MPI_File_errhandler_function(MPI_File *file, int *error_code,);</pre>	6 7
typedef	<pre>void MPI_Session_errhandler_function(MPI_Session *session,</pre>	8 9 10
typedef	<pre>int MPI_Grequest_query_function(void *extra_state,</pre>	11 12 13
typedef	<pre>int MPI_Grequest_free_function(void *extra_state);</pre>	14
typedef	<pre>int MPI_Grequest_cancel_function(void *extra_state, int complete);</pre>	15 16
typedef	<pre>int MPI_Datarep_extent_function(MPI_Datatype datatype,</pre>	17 18
typedef	<pre>int MPI_Datarep_conversion_function(void *userbuf, MPI_Datatype datatype, int count, void *filebuf, MPI_Offset position, void *extra_state);</pre>	19 20 21 22
typedef	<pre>void MPI_T_event_cb_function(MPI_T_event_instance event_instance, MPI_T_event_registration event_registration, MPI_T_cb_safety cb_safety, void *user_data);</pre>	23 24 25 26 27 28 29 30
typedef	<pre>void MPI_T_event_free_cb_function(MPI_T_event_registration event_registration, MPI_T_cb_safety cb_safety, void *user_data);</pre>	31 32 33 34 35
typedef	<pre>void MPI_T_event_dropped_cb_function(int count, MPI_T_event_registration event_registration, MPI_T_cb_safety cb_safety, void *user_data);</pre>	36 37 38 39 40
Fortran 2	008 Bindings with the mpi_f08 Module	41 42
The ABSTRAC	back prototypes when using the Fortran mpi_f08 module are shown below: user-function argument to MPI_Op_create should be declared according to: INTERFACE	43 44 45
USE	UTINE MPI_User_function(invec, inoutvec, len, datatype) , INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	46 47
TYPI	E(C_PTR), VALUE :: invec, inoutvec	48

```
1
         INTEGER :: len
2
         TYPE(MPI_Datatype) :: datatype
3
         The copy and delete function arguments to MPI_Comm_create_keyval should be de-
4
     clared according to:
5
     ABSTRACT INTERFACE
6
       SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
7
                    attribute_val_in, attribute_val_out, flag, ierror)
8
         TYPE(MPI_Comm) :: oldcomm
9
         INTEGER :: comm_keyval, ierror
10
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
11
                    attribute_val_out
12
         LOGICAL :: flag
13
14
     ABSTRACT INTERFACE
15
       SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
16
                    attribute_val, extra_state, ierror)
17
         TYPE(MPI_Comm) :: comm
18
         INTEGER :: comm_keyval, ierror
19
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
20
         The copy and delete function arguments to MPI_Win_create_keyval should be declared
21
     according to:
22
     ABSTRACT INTERFACE
23
       SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
24
                    attribute_val_in, attribute_val_out, flag, ierror)
25
         TYPE(MPI_Win) :: oldwin
26
         INTEGER :: win_keyval, ierror
27
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
28
                    attribute_val_out
29
         LOGICAL :: flag
30
31
     ABSTRACT INTERFACE
32
       SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
33
                    extra_state, ierror)
34
         TYPE(MPI_Win) :: win
35
         INTEGER :: win_keyval, ierror
36
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
37
         The copy and delete function arguments to MPI_Type_create_keyval should be declared
38
     according to:
39
     ABSTRACT INTERFACE
40
       SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
41
                    attribute_val_in, attribute_val_out, flag, ierror)
42
         TYPE(MPI_Datatype) :: oldtype
43
         INTEGER :: type_keyval, ierror
44
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
45
                    attribute_val_out
46
         LOGICAL :: flag
47
48
```

```
ABSTRACT INTERFACE
                                                                                      2
  SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
               attribute_val, extra_state, ierror)
    TYPE(MPI_Datatype) :: datatype
    INTEGER :: type_keyval, ierror
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
    The handler-function argument to MPI_Comm_create_errhandler should be declared
like this:
ABSTRACT INTERFACE
                                                                                      10
  SUBROUTINE MPI_Comm_errhandler_function(comm, error_code)
                                                                                      11
    TYPE(MPI_Comm) :: comm
                                                                                      12
    INTEGER :: error_code
                                                                                      13
                                                                                      14
    The handler-function argument to MPI_Win_create_errhandler should be declared like
                                                                                      15
this:
                                                                                      16
ABSTRACT INTERFACE
                                                                                      17
  SUBROUTINE MPI_Win_errhandler_function(win, error_code)
                                                                                      18
    TYPE(MPI_Win) :: win
                                                                                      19
    INTEGER :: error_code
                                                                                      20
    The handler-function argument to MPI_File_create_errhandler should be declared like
                                                                                      21
this:
                                                                                      22
ABSTRACT INTERFACE
                                                                                      23
  SUBROUTINE MPI_File_errhandler_function(file, error_code)
                                                                                      24
    TYPE(MPI_File) :: file
                                                                                      25
    INTEGER :: error_code
                                                                                      26
                                                                                      27
ABSTRACT INTERFACE
                                                                                      28
  SUBROUTINE MPI_File_errhandler_function(file, error_code)
                                                                                      29
    TYPE(MPI_File) :: file
                                                                                      30
    INTEGER :: error_code
                                                                                      31
    The handler-function argument to MPI_Session_create_errhandler should be declared
                                                                                      32
like this:
                                                                                      33
ABSTRACT INTERFACE
                                                                                      34
 SUBROUTINE MPI_Session_errhandler_function(session, error_code)
                                                                                      35
    TYPE(MPI_Session) :: session
                                                                                      36
    INTEGER :: error_code
                                                                                      37
                                                                                      38
ABSTRACT INTERFACE
                                                                                      39
  SUBROUTINE MPI_Session_errhandler_function(session, error_code)
                                                                                      40
    TYPE(MPI_Session) :: session
                                                                                      41
    INTEGER :: error_code
                                                                                      42
    The query, free, and cancel function arguments to MPI_Grequest_start should be de-
                                                                                      43
clared according to:
                                                                                      44
ABSTRACT INTERFACE
                                                                                      45
  SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
                                                                                      46
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
                                                                                      47
    TYPE(MPI_Status) :: status
                                                                                      48
```

```
1
         INTEGER :: ierror
\mathbf{2}
     ABSTRACT INTERFACE
3
       SUBROUTINE MPI_Grequest_free_function(extra_state, ierror)
4
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
5
         INTEGER :: ierror
6
7
     ABSTRACT INTERFACE
8
       SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
9
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
10
         LOGICAL :: complete
11
         INTEGER :: ierror
12
         The extent and conversion function arguments to MPI_Register_datarep should be de-
13
     clared according to:
14
     ABSTRACT INTERFACE
15
       SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
16
                    ierror)
17
         TYPE(MPI_Datatype) :: datatype
18
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
19
         INTEGER :: ierror
20
21
     ABSTRACT INTERFACE
22
       SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
23
                    filebuf, position, extra_state, ierror)
24
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
25
         TYPE(C_PTR), VALUE :: userbuf, filebuf
26
         TYPE(MPI_Datatype) :: datatype
27
         INTEGER :: count, ierror
28
         INTEGER(KIND=MPI_OFFSET_KIND) :: position
29
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
30
^{31}
     Fortran Bindings with mpif.h or the mpi Module
32
33
     With the Fortran mpi module or mpif.h, here are examples of how each of the user-defined
34
     subroutines should be declared.
35
         The user-function argument to MPI_OP_CREATE should be declared like this:
36
                    SUBROUTINE USER_FUNCTION (INVEC, INOUTVEC, LEN, DATATYPE)
37
         <type> INVEC(LEN), INOUTVEC(LEN)
38
         INTEGER LEN, DATATYPE
39
         The copy and delete function arguments to MPI_COMM_CREATE_KEYVAL should be
40
     declared like these:
41
     SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
42
                   ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
43
         INTEGER OLDCOMM, COMM_KEYVAL, IERROR
44
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
45
                    ATTRIBUTE_VAL_OUT
46
47
         LOGICAL FLAG
48
```

SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,	1
EXTRA_STATE, IERROR) INTEGER COMM, COMM_KEYVAL, IERROR	2 3
INTEGER COMM, COMM_REIVAL, TEMOOR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	4
	5
The copy and delete function arguments to MPI_WIN_CREATE_KEYVAL should be declared like these:	6
SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,	7
ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)	8 9
INTEGER OLDWIN, WIN_KEYVAL, IERROR	9 10
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	11
ATTRIBUTE_VAL_OUT	12
LOGICAL FLAG	13
SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,	14 15
EXTRA_STATE, IERROR) INTEGER WIN, WIN_KEYVAL, IERROR	16
INTEGER WIN, WIN_KEIVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	17
	18
The delete function argument to MPI_SESSION_CREATE_KEYVAL should be declared like this:	19
like tills:	20
SUBROUTINE SESSION_DELETE_ATTR_FUNCTION(SESSION, SESSION_KEYVAL, ATTRIBUTE_VAL EXTRA_STATE, IERROR)	21 , ₂₂
INTEGER SESSION, SESSION_KEYVAL, IERROR	23
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	24
	25 26
The copy and delete function arguments to MPI_TYPE_CREATE_KEYVAL should be	20
declared like these:	28
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)	29
INTEGER OLDTYPE, TYPE_KEYVAL, IERROR	30
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	31
ATTRIBUTE_VAL_OUT	32 33
LOGICAL FLAG	33 34
SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,	35
EXTRA_STATE, IERROR)	36
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	37
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	38
The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de-	39
clared like this:	40 41
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)	41
INTEGER COMM, ERROR_CODE	43
The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de-	44
clared like this:	45
SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)	46
INTEGER WIN, ERROR_CODE	47
	48

1	The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de-
2 3	clared like this:
4	SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE
5	INTEGER FILE, ERROR_CODE
6	The handler-function argument to $MPI_SESSION_CREATE_ERRHANDLER$ should be
7	declared like this:
8	SUBROUTINE SESSION_ERRHANDLER_FUNCTION(SESSION, ERROR_CODE)
9	INTEGER SESSION, ERROR_CODE
10	The query, free, and cancel function arguments to MPI_GREQUEST_START should be
11	declared like these:
12	SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
13	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
14	INTEGER STATUS(MPI_STATUS_SIZE), IERROR
15 16	SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
17	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
18	INTEGER IERROR
19	SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
20	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
21	LOGICAL COMPLETE
22	INTEGER IERROR
23	
24	The extent and conversion function arguments to MPI_REGISTER_DATAREP should be declared like these:
25 26	SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
20 27	INTEGER DATATYPE, IERROR
28	INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
29	SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
30	POSITION, EXTRA_STATE, IERROR)
31	<type> USERBUF(*), FILEBUF(*)</type>
32	INTEGER DATATYPE, COUNT, IERROR
33	INTEGER(KIND=MPI_OFFSET_KIND) POSITION
34	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
35	
36 37	A.1.4 Deprecated Prototype Definitions
38	
39	The following are defined C typedefs for deprecated user-defined functions, also included in
40	the file mpi.h.
41	<pre>/* prototypes for user-defined functions */</pre>
42	, procoppos for abor dorinou functions ,
43	typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
44	<pre>void *extra_state, void *attribute_val_in,</pre>
45 46	<pre>void *attribute_val_out, int *flag);</pre>
40	typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
48	<pre>void *attribute_val, void *extra_state);</pre>

1 The following are deprecated Fortran user-defined callback subroutine prototypes. The $\mathbf{2}$ deprecated copy and delete function arguments to MPI_KEYVAL_CREATE should be de-3 clared like these: SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, 4 ATTRIBUTE_VAL_OUT, FLAG, IERR) 5INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, 6 7 ATTRIBUTE_VAL_OUT, IERR LOGICAL FLAG 9 SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR) 10 INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR 11 1213 A.1.5 Info Keys 14The following info keys are reserved. They are strings. 1516"access_style" 17 "accumulate_ops" 18 "accumulate_ordering" "alloc_shared_noncontig" 1920"appnum" "arch" 21"cb_block_size" 22 "cb_buffer_size" 23"cb_nodes" 24 25"chunked_item" 26"chunked_size" "chunked" 27"collective_buffering" 28 "file" 29"file_perm" 30 31 "filename" "host" 32 "io_node_list" 33 "ip_address" 34 "ip_port" 35"mpi_assert_allow_overtaking" 36 "mpi_assert_exact_length" 37 "mpi_assert_no_any_source" 38 39 "mpi_assert_no_any_tag" "mpi_assert_strict_start_ordering" 40 41 "mpi_hw_resource_type" 42"mpi_initial_errhandler" "mpi_optimization_goal" 43 "mpi_reuse_count" 44"mpi_minimum_memory_alignment" 4546"nb_proc" "no_locks" 4748 "num_io_nodes"

```
1
      "path"
^{2}
      "same_disp_unit"
3
      "same_size"
4
      "soft"
\mathbf{5}
      "striping_factor"
6
      "striping_unit"
\overline{7}
      "wdir"
8
9
10
      A.1.6 Info Values
11
      The following info values are reserved. They are strings.
12
13
      "false"
      "mpi_errors_abort"
14
      "mpi_errors_are_fatal"
15
16
      "mpi_errors_return"
17
      "mpi_shared_memory"
18
      "random"
19
      "rar"
20
      "raw"
21
      "read_mostly"
22
      "read_once"
      "reverse_sequential"
23
^{24}
      "same_op"
25
      "same_op_no_op"
26
      "sequential"
27
      "true"
28
      "war"
29
      "waw"
      "write_mostly"
30
      "write_once"
31
32
33
              Summary of the Semantics of all Operation-Related MPI Procedures
      A.2
34
35
      A summary of the semantics of all operation-related MPI procedures can be found in [51].
36
37
38
39
40
41
42
43
44
45
46
47
```

A.3 C Bindings	1
A.3.1 Point-to-Point Communication C Bindings	3
<pre>int MPI_Bsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	4 5 6
<pre>int MPI_Bsend_init(const void *buf, int count, MPI_Datatype datatype,</pre>	7 8
<pre>int MPI_Buffer_attach(void *buffer, int size)</pre>	9 10
<pre>int MPI_Buffer_detach(void *buffer_addr, int *size)</pre>	11
int MPI_Cancel(MPI_Request *request)	12 13
<pre>int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype,</pre>	14 15
<pre>int MPI_Ibsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	16 17 18
<pre>int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag,</pre>	19 20 21
<pre>int MPI_Imrecv(void *buf, int count, MPI_Datatype datatype,</pre>	21 22 23
<pre>int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,</pre>	24 25 26
<pre>int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,</pre>	20 27 28
<pre>int MPI_Irsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	29 30 31
<pre>int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	32 33
<pre>int MPI_Isendrecv(const void *sendbuf, int sendcount,</pre>	34 35 36 37 38
<pre>int MPI_Isendrecv_replace(void *buf, int count, MPI_Datatype datatype,</pre>	39 40 41
<pre>int MPI_Issend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	42 43 44
<pre>int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message, MPI_Status *status)</pre>	45 46 47

1 int MPI_Mrecv(void *buf, int count, MPI_Datatype datatype, $\mathbf{2}$ MPI_Message *message, MPI_Status *status) 3 int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status) 4 $\mathbf{5}$ int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, 6 int tag, MPI_Comm comm, MPI_Status *status) 7 int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source, 8 int tag, MPI_Comm comm, MPI_Request *request) 9 10int MPI_Request_free(MPI_Request *request) 11 int MPI_Request_get_status(MPI_Request request, int *flag, 12MPI_Status *status) 13 14int MPI_Rsend(const void *buf, int count, MPI_Datatype datatype, int dest, 15int tag, MPI_Comm comm) 16int MPI_Rsend_init(const void *buf, int count, MPI_Datatype datatype, 17int dest, int tag, MPI_Comm comm, MPI_Request *request) 18 19int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, 20int tag, MPI_Comm comm) 21int MPI_Send_init(const void *buf, int count, MPI_Datatype datatype, 22int dest, int tag, MPI_Comm comm, MPI_Request *request) 23 24 int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, 25int dest, int sendtag, void *recvbuf, int recvcount, 26MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, 27MPI_Status *status) 28 int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, 29int dest, int sendtag, int source, int recvtag, MPI_Comm comm, 30 MPI_Status *status) 31 32 int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype, int dest, 33 int tag, MPI_Comm comm) 34int MPI_Ssend_init(const void *buf, int count, MPI_Datatype datatype, 35int dest, int tag, MPI_Comm comm, MPI_Request *request) 36 37 int MPI_Startall(int count, MPI_Request array_of_requests[]) 38 39int MPI_Start(MPI_Request *request) 40int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag, 41 MPI_Status array_of_statuses[]) 4243int MPI_Testany(int count, MPI_Request array_of_requests[], int *index, 44int *flag, MPI_Status *status) 45int MPI_Test_cancelled(const MPI_Status *status, int *flag) 4647int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status) 48

<pre>int MPI_Testsome(int incount, MPI_Request array_of_requests[],</pre>	1
<pre>int *outcount, int array_of_indices[],</pre>	2
<pre>MPI_Status array_of_statuses[])</pre>	3
<pre>int MPI_Waitall(int count, MPI_Request array_of_requests[],</pre>	4
MPI_Status array_of_statuses[])	5 6
int MDI Waitany(int count MDI Request arroy of requests[] int tinder	7
<pre>int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, MPI_Status *status)</pre>	8
	9
<pre>int MPI_Wait(MPI_Request *request, MPI_Status *status)</pre>	10
<pre>int MPI_Waitsome(int incount, MPI_Request array_of_requests[],</pre>	11
<pre>int *outcount, int array_of_indices[],</pre>	12
<pre>MPI_Status array_of_statuses[])</pre>	13
	14 15
A.3.2 Partitioned Communication C Bindings	16
J	17
<pre>int MPI_Parrived(MPI_Request *request, int partition, int *flag)</pre>	18
<pre>int MPI_Pready(int partition, MPI_Request *request)</pre>	19
<pre>int MPI_Pready_list(int length, int array_of_partitions[],</pre>	20
MPI_Request *request)	21
	22
<pre>int MPI_Pready_range(int partition_low, int partition_high,</pre>	23 24
MPI_Request *request)	24 25
<pre>int MPI_Precv_init(void *buf, int partitions, MPI_Count count,</pre>	26
MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,	27
MPI_Info info, MPI_Request *request)	28
int MPI_Psend_init(void *buf, int partitions, MPI_Count count,	29
MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,	30
MPI_Info info, MPI_Request *request)	31
	32
A.3.3 Datatypes C Bindings	33 34
	35
<pre>int MPI_Get_address(const void *location, MPI_Aint *address)</pre>	36
<pre>int MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,</pre>	37
int *count)	38
int MPI_Get_elements_x(const MPI_Status *status, MPI_Datatype datatype,	39
MPI_Count *count)	40
int MDT Dash antanal (and she datawa [] and said tinhaf int income	41
<pre>int MPI_Pack_external(const char datarep[], const void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,</pre>	42
MPI_Aint *position)	43 44
•	45
int MPI_Pack_external_size(const char datarep[], int incount,	46
MPI_Datatype datatype, MPI_Aint *size)	47
	48

1int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype, $\mathbf{2}$ void *outbuf, int outsize, int *position, MPI_Comm comm) 3 int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, 4 int *size) 56 int MPI_Type_commit(MPI_Datatype *datatype) 7 int MPI_Type_contiguous(int count, MPI_Datatype oldtype, 8 MPI_Datatype *newtype) 9 10int MPI_Type_create_darray(int size, int rank, int ndims, 11 const int array_of_gsizes[], const int array_of_distribs[], 12const int array_of_dargs[], const int array_of_psizes[], 13 int order, MPI_Datatype oldtype, MPI_Datatype *newtype) 14int MPI_Type_create_hindexed_block(int count, int blocklength, 15const MPI_Aint array_of_displacements[], MPI_Datatype oldtype, 16MPI_Datatype *newtype) 1718int MPI_Type_create_hindexed(int count, const int array_of_blocklengths[], 19const MPI_Aint array_of_displacements[], MPI_Datatype oldtype, 20MPI_Datatype *newtype) 21int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride, 22 MPI_Datatype oldtype, MPI_Datatype *newtype) 2324int MPI_Type_create_indexed_block(int count, int blocklength, 25const int array_of_displacements[], MPI_Datatype oldtype, 26MPI_Datatype *newtype) 27int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, 28MPI_Aint extent, MPI_Datatype *newtype) 29 30 int MPI_Type_create_struct(int count, const int array_of_blocklengths[], 31 const MPI_Aint array_of_displacements[], 32 const MPI_Datatype array_of_types[], MPI_Datatype *newtype) 33 int MPI_Type_create_subarray(int ndims, const int array_of_sizes[], 34const int array_of_subsizes[], const int array_of_starts[], 35 int order, MPI_Datatype oldtype, MPI_Datatype *newtype) 36 37 int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype) 38 int MPI_Type_free(MPI_Datatype *datatype) 39 40int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers, 41 int max_addresses, int max_datatypes, int array_of_integers[], 42MPI_Aint array_of_addresses[], 43 MPI_Datatype array_of_datatypes[]) 4445int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers, 46int *num_addresses, int *num_datatypes, int *combiner) 47int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, 48

	MPI_Aint *extent)	1
int MPI_Type	_get_extent_x(MPI_Datatype datatype, MPI_Count *1b,	2 3
• -	MPI_Count *extent)	4
int MPT Type	_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,	5
	MPI_Aint *true_extent)	6
int MDT Turne	_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_1b,	7
inc mi_iype	MPI_Count *true_extent)	8 9
· · · · MDT T-····		9 10
int MPI_Type	<pre>_indexed(int count, const int array_of_blocklengths[],</pre>	11
	MPI_Datatype *newtype)	12
int MDT Trong		13
int MPI_Type	_size(MPI_Datatype datatype, int *size)	14
int MPI_Type	_size_x(MPI_Datatype datatype, MPI_Count *size)	15 16
int MPI_Type	_vector(int count, int blocklength, int stride,	17
	MPI_Datatype oldtype, MPI_Datatype *newtype)	18
int MPI Unpa	<pre>.ck_external(const char datarep[], const void *inbuf,</pre>	19
_ 1	MPI_Aint insize, MPI_Aint *position, void *outbuf,	20
	int outcount, MPI_Datatype datatype)	21 22
int MPI_Unpa	ck(const void *inbuf, int insize, int *position, void *outbuf,	22
- 1	int outcount, MPI_Datatype datatype, MPI_Comm comm)	24
MPI Aint MPI	_Aint_add(MPI_Aint base, MPI_Aint disp)	25
		26
MPI_Aint MPI	_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)	27 28
		20
A.3.4 Collec	tive Communication C Bindings	30
int MPI_Allg	ather(const void *sendbuf, int sendcount,	31
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	32
	MPI_Datatype recvtype, MPI_Comm comm)	33
int MPI_Allg	ather_init(const void *sendbuf, int sendcount,	34 35
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	36
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,	37
	MPI_Request *request)	38
int MPI_Allg	atherv(const void *sendbuf, int sendcount,	39
	MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],	40 41
	<pre>const int displs[], MPI_Datatype recvtype, MPI_Comm comm)</pre>	41 42
int MPI_Allg	atherv_init(const void *sendbuf, int sendcount,	43
	<pre>MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],</pre>	44
	const int displs[], MPI_Datatype recvtype, MPI_Comm comm,	45
	MPI_Info info, MPI_Request *request)	46
int MPI_Allr	educe(const void *sendbuf, void *recvbuf, int count,	47 48
		40

1MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) $\mathbf{2}$ int MPI_Allreduce_init(const void *sendbuf, void *recvbuf, int count, 3 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 4 MPI_Info info, MPI_Request *request) 56 int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, 7 void *recvbuf, int recvcount, MPI_Datatype recvtype, 8 MPI_Comm comm) 9 int MPI_Alltoall_init(const void *sendbuf, int sendcount, 10 MPI_Datatype sendtype, void *recvbuf, int recvcount, 11 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 12MPI_Request *request) 13 14int MPI_Alltoallv(const void *sendbuf, const int sendcounts[], 15const int sdispls[], MPI_Datatype sendtype, void *recvbuf, 16const int recvcounts[], const int rdispls[], 17MPI_Datatype recvtype, MPI_Comm comm) 18 int MPI_Alltoallv_init(const void *sendbuf, const int sendcounts[], 19 const int sdispls[], MPI_Datatype sendtype, void *recvbuf, 20const int recvcounts[], const int rdispls[], 21MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 22 MPI_Request *request) 2324int MPI_Alltoallw(const void *sendbuf, const int sendcounts[], 25const int sdispls[], const MPI_Datatype sendtypes[], 26void *recvbuf, const int recvcounts[], const int rdispls[], 27const MPI_Datatype recvtypes[], MPI_Comm comm) 28int MPI_Alltoallw_init(const void *sendbuf, const int sendcounts[], 29 const int sdispls[], const MPI_Datatype sendtypes[], 30 void *recvbuf, const int recvcounts[], const int rdispls[], 31const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info, 32 MPI_Request *request) 33 34 int MPI_Barrier(MPI_Comm comm) 35 int MPI_Barrier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request) 36 37 int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, 38 MPI_Comm comm) 39 int MPI_Bcast_init(void *buffer, int count, MPI_Datatype datatype, 40int root, MPI_Comm comm, MPI_Info info, MPI_Request *request) 41 42int MPI_Exscan(const void *sendbuf, void *recvbuf, int count, 43 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) 44 int MPI_Exscan_init(const void *sendbuf, void *recvbuf, int count, 45MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 46MPI_Info info, MPI_Request *request) 47 48

int MPI_Gath	er(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	1 2 3
int MPI_Gath	er_init(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)	4 5 6 7 8
int MPI_Gath	erv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm)	9 10 11 12
int MPI_Gath	erv_init(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)	13 14 15 16
int MPI_Iall	gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	17 18 19 20
int MPI_Iall	<pre>gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	20 21 22 23 24
int MPI_Iall	reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)	25 26 27 28
int MPI_Iall	<pre>toal1(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	29 30 31
int MPI_Iall	<pre>toallv(const void *sendbuf, const int sendcounts[], const int sdispls[], MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	32 33 34 35 36
int MPI_Iall	<pre>toallw(const void *sendbuf, const int sendcounts[], const int sdispls[], const MPI_Datatype sendtypes[], void *recvbuf, const int recvcounts[], const int rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Request *request)</pre>	37 38 39 40 41
int MPI_Ibar	rier(MPI_Comm comm, MPI_Request *request)	42 43
int MPI_Ibca	st(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm, MPI_Request *request)	44 45
int MPI_Iexs	can(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	46 47 48

1		MPI_Request *request)
2	int	MPI_Igather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
3	1110	void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
4		MPI_Comm comm, MPI_Request *request)
5 6		
7	int	MPI_Igatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
8		<pre>void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm,</pre>
9		MPI_Batatype recorype, int root, MPI_comm comm, MPI_Request *request)
10		
11	int	MPI_Ireduce(const void *sendbuf, void *recvbuf, int count,
12		MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
13		MPI_Request *request)
14	int	<pre>MPI_Ireduce_scatter_block(const void *sendbuf, void *recvbuf,</pre>
15		<pre>int recvcount, MPI_Datatype datatype, MPI_Op op,</pre>
16 17		MPI_Comm comm, MPI_Request *request)
18	int	MPI_Ireduce_scatter(const void *sendbuf, void *recvbuf,
19		<pre>const int recvcounts[], MPI_Datatype datatype, MPI_Op op,</pre>
20		MPI_Comm comm, MPI_Request *request)
21	int	MPI_Iscan(const void *sendbuf, void *recvbuf, int count,
22	1110	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
23		MPI_Request *request)
24		MPI_Iscatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
25	THE	void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
26 27		MPI_Comm comm, MPI_Request *request)
28		
29	int	<pre>MPI_Iscatterv(const void *sendbuf, const int sendcounts[],</pre>
30		<pre>const int displs[], MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,</pre>
31		MPI_Request *request)
32		
33	int	<pre>MPI_Op_commutative(MPI_Op op, int *commute)</pre>
34 35	int	<pre>MPI_Op_create(MPI_User_function *user_fn, int commute, MPI_Op *op)</pre>
36	int	MPI_Op_free(MPI_Op *op)
37	int	MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
38	1110	MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
39		
40	int	MPI_Reduce_init(const void *sendbuf, void *recvbuf, int count,
41 42		MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
42		MPI_Info info, MPI_Request *request)
44	int	<pre>MPI_Reduce_local(const void *inbuf, void *inoutbuf, int count,</pre>
45		MPI_Datatype datatype, MPI_Op op)
46	int	MPI_Reduce_scatter_block(const void *sendbuf, void *recvbuf,
47		int recvcount, MPI_Datatype datatype, MPI_Op op,
48		MPI_Comm comm)

int	<pre>MPI_Reduce_scatter_block_init(const void *sendbuf, void *recvbuf, int recvcount, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>	1 2 3
int	<pre>MPI_Reduce_scatter(const void *sendbuf, void *recvbuf,</pre>	4 5 6 7
int	<pre>MPI_Reduce_scatter_init(const void *sendbuf, void *recvbuf, const int recvcounts[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>	8 9 10 11
int	MPI_Scan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	12 13
int	MPI_Scan_init(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request)	14 15 16 17
int	<pre>MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)</pre>	18 19 20
int	<pre>MPI_Scatter_init(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>	21 22 23 24 25
int	<pre>MPI_Scatterv(const void *sendbuf, const int sendcounts[],</pre>	23 26 27 28
int	<pre>MPI_Scatterv_init(const void *sendbuf, const int sendcounts[],</pre>	29 30 31 32 33 34
A.3.	5 Groups, Contexts, Communicators, and Caching C Bindings	35 36
int	MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)	37
int	MPI_Comm_create_from_group(MPI_Group group, const char *stringtag, MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newcomm)	38 39
int	MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)	40 41
int	MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag, MPI_Comm *newcomm)	42 43
int	<pre>MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,</pre>	44 45 46 47 48

ANNEX A. LANGUAGE BINDINGS SUMMARY

1	int	MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)
2 3 4	int	<pre>MPI_COMM_DUP_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
5	int	MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
6 7	int	MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
8 9	int	MPI_Comm_free(MPI_Comm *comm)
9 10	int	MPI_Comm_free_keyval(int *comm_keyval)
11 12 13	int	<pre>MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,</pre>
14	int	MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)
15 16	int	<pre>MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)</pre>
17	int	MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
18 19	int	MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)
20 21	int	<pre>MPI_Comm_idup_with_info(MPI_Comm comm, MPI_Info info,</pre>
22 23 24 25	int	<pre>MPI_COMM_NULL_COPY_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
26 27	int	<pre>MPI_COMM_NULL_DELETE_FN(MPI_Comm comm, int comm_keyval,</pre>
28 29	int	MPI_Comm_rank(MPI_Comm comm, int *rank)
30	int	MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
31 32	int	<pre>MPI_Comm_remote_size(MPI_Comm comm, int *size)</pre>
33	int	MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)
34 35	int	MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)
36 37	int	MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)
38	int	MPI_Comm_size(MPI_Comm comm, int *size)
39 40	int	MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
41 42	int	<pre>MPI_Comm_split_type(MPI_Comm comm, int split_type, int key, MPI_Info info, MPI_Comm *newcomm)</pre>
43 44	int	<pre>MPI_Comm_test_inter(MPI_Comm comm, int *flag)</pre>
45	int	<pre>MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)</pre>
46 47 48	int	<pre>MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)</pre>

<pre>int MPI_Group_excl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)</pre>	1 2
<pre>int MPI_Group_free(MPI_Group *group)</pre>	3
	4
<pre>int MPI_Group_from_session_pset(MPI_Session session, const char *pset_name,</pre>	5 6
<pre>int MPI_Group_incl(MPI_Group group, int n, const int ranks[],</pre>	7
MPI_Group *newgroup)	8 9
int MDI Group intergration (MDI Group group1 MDI Group group2	10
<pre>int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)</pre>	11
<pre>int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],</pre>	12 13
MPI_Group *newgroup)	14
<pre>int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],</pre>	15 16
MPI_Group *newgroup)	10
<pre>int MPI_Group_rank(MPI_Group group, int *rank)</pre>	18
<pre>int MPI_Group_size(MPI_Group group, int *size)</pre>	19
	20
<pre>int MPI_Group_translate_ranks(MPI_Group group1, int n, const int ranks1[],</pre>	21
<pre>MPI_Group group2, int ranks2[])</pre>	22
<pre>int MPI_Group_union(MPI_Group group1, MPI_Group group2,</pre>	23
MPI_Group *newgroup)	24 25
<pre>int MPI_Intercomm_create_from_groups(MPI_Group local_group,</pre>	26
int local_leader, MPI_Group remote_group, int remote_leader,	27
const char *stringtag, MPI_Info info,	28
MPI_Errhandler errhandler, MPI_Comm *newintercomm)	29
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,	30
MPI_Comm peer_comm, int remote_leader, int tag,	31
MPI_Comm *newintercomm)	32
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,	33 34
MPI_Comm *newintracomm)	35
	36
<pre>int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,</pre>	37
<pre>MPI_Type_delete_attr_function *type_delete_attr_fn,</pre>	38
<pre>int *type_keyval, void *extra_state)</pre>	39
<pre>int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)</pre>	40
int MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval,	41
void *extra_state, void *attribute_val_in,	42
void *attribute_val_out, int *flag)	43 44
int MPI_Type_free_keyval(int *type_keyval)	44
	46
<pre>int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval,</pre>	47
<pre>void *attribute_val, int *flag)</pre>	48

1 2	<pre>int MPI_Type_get_name(MPI_Datatype datatype, char *type_name,</pre>
3 4 5 6	<pre>int MPI_TYPE_NULL_COPY_FN(MPI_Datatype oldtype, int type_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
7 8	<pre>int MPI_TYPE_NULL_DELETE_FN(MPI_Datatype datatype, int type_keyval,</pre>
9 10 11	<pre>int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,</pre>
12	int MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)
13 14 15 16	<pre>int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,</pre>
17 18	int MPI_Win_delete_attr(MPI_Win win, int win_keyval)
19 20	<pre>int MPI_WIN_DUP_FN(MPI_Win oldwin, int win_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
21 22	int MPI_Win_free_keyval(int *win_keyval)
23 24	<pre>int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,</pre>
25 26	int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
27 28	<pre>int MPI_WIN_NULL_COPY_FN(MPI_Win oldwin, int win_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
29 30 31	<pre>int MPI_WIN_NULL_DELETE_FN(MPI_Win win, int win_keyval,</pre>
32	int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
33 34 35	<pre>int MPI_Win_set_name(MPI_Win win, const char *win_name)</pre>
36	A.3.6 Process Topologies C Bindings
37 38	int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
39 40	<pre>int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],</pre>
41 42	int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
43 44	<pre>int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],</pre>
45 46 47 48	<pre>int MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],</pre>

<pre>int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank) int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,</pre>			
<pre>int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,</pre>	int MPI_Car	rt_rank(MPI_Comm comm, const int coords[], int *rank)	1
<pre>int *rank_source, int *rank_dest) int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm) int MPI_Dims_create(int nnodes, int ndims, int dims[]) int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,</pre>	int MPI Car	rt shift(MPI Comm comm, int direction, int disp.	
<pre>int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm) int MPI_Dims_create(int nnodes, int ndims, int dims[]) int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,</pre>		-	
<pre>int MPI_Dims_create(int nnodes, int ndims, int dims[]) int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,</pre>	int MPT Car	t sub(MPI Comm comm const int remain dims[] MPI Comm *newcomm)	
<pre>int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,</pre>			6
<pre>int MPI_DISE_graph_create_adjacent(WPI_Comm comm_oid, int lindegree,</pre>	int MPI_Dim	ns_create(int nnodes, int ndims, int dims[])	7
<pre>const int sources[], const int sourceweights[], int outdegree,</pre>	int MPI_Dis	st_graph_create_adjacent(MPI_Comm comm_old, int indegree,	
<pre>const int destinations[], const int destiveigns[], MPI_Info info, int reorder, MPI_Comm *comm_dist_graph) int MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[], const int degrees[], const int destinations[], const int weights[], MPI_Info info, int reorder, MPI_Comm *comm_dist_graph) int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree, int *outdegree, int *weighted) int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[], int sourceweights[], int maxoutdegree, int destinations[], int destweights[]) int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[], const int edges[], int reorder, MPI_Comm *comm_graph) int MPI_Graph_det(MPI_Comm comm, int *nnodes, int *nedges) int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[], int edges[]) int MPI_Graph_map(MPI_Comm comm, int rank, int *nneighbors) int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors) int MPI_Graph_neighbors(MPI_Comm comm, int rank, int *nneighbors) int MPI_Graph_neighbors(MPI_Comm comm, int rank, int *nneighbors) int MPI_Graph_neighbors(MPI_Comm comm, int rank, int sendcount, MPI_Datatype sendtype, void *recvbuf, int reovcount, MPI_Datatype sendtype, void *recvbuf, const int reovcounts[], const int displs[], MPI_Datatype reovtype, MPI_Comm comm, 42</pre>			
<pre>MPI_INFO INFO, INT FeOrder, MPI_Comm *comm_dist_grapn) int MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[],</pre>			
<pre>const int degrees[], const int destinations[], const int weights[], MPI_Info info, int reorder, MPI_Comm *comm_dist_graph) int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree, int *outdegree, int *weighted) int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[], int sourceweights[], int maxoutdegree, int destinations[], int destweights[]) int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[], const int edges[], int reorder, MPI_Comm *comm_graph) int MPI_Graph_get(MPI_Comm comm, int *nnodes, int *nedges) int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[], int edges[]) int MPI_Graph_map(MPI_Comm comm, int rankdex, int maxedges, int index[], const int edges[], int *newrank) int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors) int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors) int MPI_Graph_neighbors(MPI_Comm comm, int rank, int *nneighbors) int MPI_Graph_neighbors(MPI_Comm comm, int rank, int *nneighbors) int MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 42</pre>		MP1_Info info, int reorder, MP1_Comm *comm_dist_graph)	
<pre>const int weights[], MPI_Info info, int reorder, MPI_Comm *comm_dist_graph) int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree, int *outdegree, int *weighted) int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[], int sourceweights[], int maxoutdegree, int destinations[], int destweights[]) int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[], const int edges[], int reorder, MPI_Comm *comm_graph) int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges) int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[], int edges[]) int MPI_Graph_neg(MPI_Comm comm, int nnodes, const int index[], const int edges[], int *newrank) int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors) int MPI_Graph_neighbors[]) int MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, int rank, int *nneighbors]</pre>	int MPI_Dis		13
<pre>MPI_Comm *comm_dist_graph) int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,</pre>		-	14
<pre>int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,</pre>			15
<pre>int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,</pre>		MP1_comm_dist_graph)	
<pre>int *outdegree, int *weighted) int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],</pre>	int MPI_Dis		
<pre>int sourceweights[], int maxoutdegree, int destinations[], int destweights[]) int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[], const int edges[], int reorder, MPI_Comm *comm_graph) int MPI_Graph_get(MPI_Comm comm, int *nnodes, int *nedges) int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[], int edges[]) int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[], const int edges[], int *newrank) int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors) int MPI_Graph_neighbors(MPI_Comm comm, int rank, int *nneighbors) int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int neighbors[]) int MPI_Ineighbor_allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcounts, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 42</pre>		int *outdegree, int *weighted)	
<pre>int destweights[]) 22 int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],</pre>	int MPI_Dis	st_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],	20
<pre>int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],</pre>			21
<pre>int MP1_Graph_create(MP1_Comm comm_old, int nnodes, const int index[],</pre>		int destweights[])	22
<pre>const int edges[], int reorder, MPI_Comm *comm_graph) int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges) int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[],</pre>	int MPI_Gra	aph_create(MPI_Comm comm_old, int nnodes, const int index[],	
<pre>int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges) 26 int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[], 27 int edges[]) 29 int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[], 30 const int edges[], int *newrank) 31 int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors) 33 int MPI_Graph_neighbors(MPI_Comm comm, int rank, int *nneighbors) 33 int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, 34 int neighbors[]) 35 int MPI_Ineighbor_allgather(const void *sendbuf, int sendcount, 37 MPI_Datatype sendtype, void *recvbuf, int recvcount, 38 int MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount, 39 int MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], 29 int MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], 41 const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 42</pre>		<pre>const int edges[], int reorder, MPI_Comm *comm_graph)</pre>	
<pre>int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[],</pre>	int MPI_Gra	aphdims_get(MPI_Comm comm, int *nnodes, int *nedges)	
<pre>int edges[]) int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[],</pre>	int MPI_Gra	aph_get(MPI_Comm comm, int maxindex, int maxedges, int index[],	27
<pre>int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[],</pre>			
<pre>const int edges[], int *newrank) 31 int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors) 33 int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, 34 int neighbors[]) 35 int MPI_Ineighbor_allgather(const void *sendbuf, int sendcount, 37 MPI_Datatype sendtype, void *recvbuf, int recvcount, 38 MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 39 int MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount, 40 MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], 41 const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 42</pre>	int MPT Gra	aph map(MPT Comm comm int prodes const int index[]	
<pre>int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors) int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,</pre>	1110 111 1_010		
<pre>int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,</pre>	int MDT Core		32
<pre>int neighbors[]) 35 int MPI_Ineighbor_allgather(const void *sendbuf, int sendcount,</pre>	int MP1_Gra	aph_heighbors_count(MP1_Comm comm, int rank, int *hneighbors)	33
<pre>36 int MPI_Ineighbor_allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 39 int MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 42</pre>	int MPI_Gra		
<pre>int MPI_Ineighbor_allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) int MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 42</pre>		int neighbors[])	
<pre>MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) int MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm,</pre>	int MPI_Ine	ighbor_allgather(const void *sendbuf, int sendcount,	
<pre>int MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm,</pre>			
MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 42		MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	39
const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 42	int MPI_Ine		40
MDT D = m = n + 1 + m = n + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 +			
MPI_Request *request) 43		MP1_kequest *request)	
int MPI_Ineighbor_alltoall(const void *sendbuf, int sendcount, $_{45}$	int MPI_Ine		
MPI_Datatype sendtype, void *recvbuf, int recvcount,			46
MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 47		MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	47
48			48

```
1
     int MPI_Ineighbor_alltoallv(const void *sendbuf, const int sendcounts[],
\mathbf{2}
                   const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
3
                   const int recvcounts[], const int rdispls[],
4
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
5
     int MPI_Ineighbor_alltoallw(const void *sendbuf, const int sendcounts[],
6
                   const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
7
                   void *recvbuf, const int recvcounts[],
8
                   const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
9
                   MPI_Comm comm, MPI_Request *request)
10
11
     int MPI_Neighbor_allgather(const void *sendbuf, int sendcount,
12
                   MPI_Datatype sendtype, void *recvbuf, int recvcount,
13
                   MPI_Datatype recvtype, MPI_Comm comm)
14
     int MPI_Neighbor_allgather_init(const void *sendbuf, int sendcount,
15
                   MPI_Datatype sendtype, void *recvbuf, int recvcount,
16
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
17
                   MPI_Request *request)
18
19
     int MPI_Neighbor_allgatherv(const void *sendbuf, int sendcount,
20
                   MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
21
                   const int displs[], MPI_Datatype recvtype, MPI_Comm comm)
22
     int MPI_Neighbor_allgatherv_init(const void *sendbuf, int sendcount,
23
                   MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
24
                   const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
25
                   MPI_Info info, MPI_Request *request)
26
27
     int MPI_Neighbor_alltoall(const void *sendbuf, int sendcount,
28
                   MPI_Datatype sendtype, void *recvbuf, int recvcount,
29
                   MPI_Datatype recvtype, MPI_Comm comm)
30
     int MPI_Neighbor_alltoall_init(const void *sendbuf, int sendcount,
^{31}
                   MPI_Datatype sendtype, void *recvbuf, int recvcount,
32
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
33
                   MPI_Request *request)
34
35
     int MPI_Neighbor_alltoallv(const void *sendbuf, const int sendcounts[],
36
                   const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
37
                   const int recvcounts[], const int rdispls[],
38
                  MPI_Datatype recvtype, MPI_Comm comm)
39
     int MPI_Neighbor_alltoallv_init(const void *sendbuf,
40
                   const int sendcounts[], const int sdispls[],
41
                   MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
42
                   const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm,
43
                   MPI_Info info, MPI_Request *request)
44
45
     int MPI_Neighbor_alltoallw(const void *sendbuf, const int sendcounts[],
46
                   const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
47
                   void *recvbuf, const int recvcounts[],
48
```

```
1
              const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
                                                                                   2
             MPI_Comm comm)
                                                                                   3
int MPI_Neighbor_alltoallw_init(const void *sendbuf,
             const int sendcounts[], const MPI_Aint sdispls[],
                                                                                   5
              const MPI_Datatype sendtypes[], void *recvbuf,
                                                                                   6
              const int recvcounts[], const MPI_Aint rdispls[],
              const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
                                                                                   8
             MPI_Request *request)
                                                                                   9
                                                                                   10
int MPI_Topo_test(MPI_Comm comm, int *status)
                                                                                   11
                                                                                   12
A.3.7 MPI Environmental Management C Bindings
                                                                                   13
                                                                                   14
int MPI_Add_error_class(int *errorclass)
                                                                                   15
int MPI_Add_error_code(int errorclass, int *errorcode)
                                                                                   16
                                                                                   17
int MPI_Add_error_string(int errorcode, const char *string)
                                                                                   18
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
                                                                                   19
                                                                                   20
int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)
                                                                                   21
int MPI_Comm_create_errhandler(
                                                                                   22
             MPI_Comm_errhandler_function *comm_errhandler_fn,
                                                                                   23
             MPI_Errhandler *errhandler)
                                                                                   ^{24}
                                                                                   25
int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
                                                                                   26
int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
                                                                                   27
                                                                                   28
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
                                                                                   29
int MPI_Error_class(int errorcode, int *errorclass)
                                                                                   30
                                                                                   31
int MPI_Error_string(int errorcode, char *string, int *resultlen)
                                                                                   32
                                                                                   33
int MPI_File_call_errhandler(MPI_File fh, int errorcode)
                                                                                   34
int MPI_File_create_errhandler(
                                                                                   35
             MPI_File_errhandler_function *file_errhandler_fn,
                                                                                   36
             MPI_Errhandler *errhandler)
                                                                                   37
                                                                                   38
int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
                                                                                   39
int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
                                                                                   40
                                                                                   41
int MPI_Free_mem(void *base)
                                                                                   42
int MPI_Get_library_version(char *version, int *resultlen)
                                                                                   43
                                                                                   44
int MPI_Get_processor_name(char *name, int *resultlen)
                                                                                   45
int MPI_Get_version(int *version, int *subversion)
                                                                                   46
                                                                                   47
int MPI_Session_call_errhandler(MPI_Session session, int errorcode)
                                                                                   48
```

```
1
     int MPI_Session_create_errhandler(
\mathbf{2}
                   MPI_Session_errhandler_function *session_errhandler_fn,
3
                   MPI_Errhandler *errhandler)
4
     int MPI_Session_get_errhandler(MPI_Session session,
5
                   MPI_Errhandler *errhandler)
6
\overline{7}
     int MPI_Session_set_errhandler(MPI_Session session,
8
                   MPI_Errhandler errhandler)
9
     int MPI_Win_call_errhandler(MPI_Win win, int errorcode)
10
11
     int MPI_Win_create_errhandler(
12
                   MPI_Win_errhandler_function *win_errhandler_fn,
13
                   MPI_Errhandler *errhandler)
14
     int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
15
16
     int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
17
     double MPI_Wtick(void)
18
19
     double MPI_Wtime(void)
20
21
     A.3.8 The Info Object C Bindings
22
23
     int MPI_Info_create_env(int argc, char argv[], MPI_Info *info)
24
25
     int MPI_Info_create(MPI_Info *info)
26
     int MPI_Info_delete(MPI_Info info, const char *key)
27
28
     int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
29
     int MPI_Info_free(MPI_Info *info)
30
31
     int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,
32
                   int *flag)
33
     int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
34
35
     int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
36
     int MPI_Info_get_string(MPI_Info info, const char *key, int *buflen,
37
                   char *value, int *flag)
38
39
     int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
40
                   int *flag)
41
     int MPI_Info_set(MPI_Info info, const char *key, const char *value)
42
43
44
     A.3.9 Process Creation and Management C Bindings
45
46
     int MPI_Abort(MPI_Comm comm, int errorcode)
47
     int MPI_Close_port(const char *port_name)
48
```

int	<pre>MPI_Comm_accept(const char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)</pre>	1 2
int	<pre>MPI_Comm_connect(const char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)</pre>	3 4 5
int	MPI_Comm_disconnect(MPI_Comm *comm)	6
int	MPI_Comm_get_parent(MPI_Comm *parent)	7 8
int	MPI_Comm_join(int fd, MPI_Comm *intercomm)	9
int	MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,	10 11
	MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])	12 13
int	MPI_Comm_spawn_multiple(int count, char *array_of_commands[],	14
	<pre>char **array_of_argv[], const int array_of_maxprocs[],</pre>	15
	<pre>const MPI_Info array_of_info[], int root, MPI_Comm comm,</pre>	16 17
	<pre>MPI_Comm *intercomm, int array_of_errcodes[])</pre>	18
int	MPI_Finalized(int *flag)	19
int	MPI_Finalize(void)	20
		21
int	<pre>MPI_Init(int *argc, char ***argv)</pre>	22
int	MPI_Initialized(int *flag)	23 24
int	MPI_Init_thread(int *argc, char ***argv, int required, int *provided)	25
int	MPI_Is_thread_main(int *flag)	26
		27
int	<pre>MPI_Lookup_name(const char *service_name, MPI_Info info,</pre>	28 29
int	MPI_Open_port(MPI_Info info, char *port_name)	30 31
int	MPI_Publish_name(const char *service_name, MPI_Info info,	32
	const char *port_name)	33
int	MPI_Query_thread(int *provided)	34
		35
int	MPI_Session_finalize(MPI_Session *session)	36 37
int	MPI_Session_get_info(MPI_Session session, MPI_Info *info_used)	38
int	<pre>MPI_Session_get_nth_pset(MPI_Session session, MPI_Info info, int n,</pre>	39 40
int	<pre>MPI_Session_get_num_psets(MPI_Session session, MPI_Info info,</pre>	41 42 43
int	<pre>MPI_Session_get_pset_info(MPI_Session session, const char *pset_name, MPI_Info *info)</pre>	43 44 45
.		46
TUL	<pre>MPI_Session_init(MPI_Info info, MPI_Errhandler errhandler, MPI_Session *session)</pre>	47 48

	000	ANNEX A. LANGUAGE DINDINGS SUMMARI
1 2 3	int MPI_Un	<pre>publish_name(const char *service_name, MPI_Info info,</pre>
4 5	A.3.10 On	e-Sided Communications C Bindings
6 7 8 9 10	int MPI_Ac	<pre>cumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)</pre>
11 12 13	int MPI_Co	<pre>mpare_and_swap(const void *origin_addr, const void *compare_addr, void *result_addr, MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Win win)</pre>
14 15 16 17	int MPI_Fe	tch_and_op(const void *origin_addr, void *result_addr, MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Op op, MPI_Win win)
18 19 20 21 22	int MPI_Ge	<pre>t_accumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, void *result_addr, int result_count, MPI_Datatype result_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)</pre>
23 24 25 26 27	int MPI_Ge	t(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)
28 29 30 31	int MPI_Pu	t(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)
32 33 34 35 36 37	int MPI_Ra	<pre>ccumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win, MPI_Request *request)</pre>
 38 39 40 41 42 43 	int MPI_Rg	<pre>et_accumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, void *result_addr, int result_count, MPI_Datatype result_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win, MPI_Request *request)</pre>
44 45 46 47 48	int MPI_Rg	et(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win,

1 MPI_Request *request) $\mathbf{2}$ int MPI_Rput(const void *origin_addr, int origin_count, 3 MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, 5 MPI_Datatype target_datatype, MPI_Win win, 6 MPI_Request *request) int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info, 9 MPI_Comm comm, void *baseptr, MPI_Win *win) 10 int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info, 11 MPI_Comm comm, void *baseptr, MPI_Win *win) 1213 int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size) 14int MPI_Win_complete(MPI_Win win) 1516int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win) 17int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, 18 MPI_Comm comm, MPI_Win *win) 19 20int MPI_Win_detach(MPI_Win win, const void *base) 21int MPI_Win_fence(int assert, MPI_Win win) 22 23int MPI_Win_flush_all(MPI_Win win) 24int MPI_Win_flush(int rank, MPI_Win win) 2526int MPI_Win_flush_local_all(MPI_Win win) 27int MPI_Win_flush_local(int rank, MPI_Win win) 28 29 int MPI_Win_free(MPI_Win *win) 30 int MPI_Win_get_group(MPI_Win win, MPI_Group *group) 3132 int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used) 33 34 int MPI_Win_lock_all(int assert, MPI_Win win) 35 int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win) 36 37 int MPI_Win_post(MPI_Group group, int assert, MPI_Win win) 38 int MPI_Win_set_info(MPI_Win win, MPI_Info info) 39 40 int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size, 41 int *disp_unit, void *baseptr) 42int MPI_Win_start(MPI_Group group, int assert, MPI_Win win) 43 44 int MPI_Win_sync(MPI_Win win) 45int MPI_Win_test(MPI_Win win, int *flag) 4647int MPI_Win_unlock_all(MPI_Win win) 48

```
1
     int MPI_Win_unlock(int rank, MPI_Win win)
\mathbf{2}
     int MPI_Win_wait(MPI_Win win)
3
4
\mathbf{5}
     A.3.11 External Interfaces C Bindings
6
     int MPI_Grequest_complete(MPI_Request request)
7
8
     int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
9
                   MPI_Grequest_free_function *free_fn,
10
                   MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
11
                   MPI_Request *request)
12
     int MPI_Status_set_cancelled(MPI_Status *status, int flag)
13
14
     int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
15
                   int count)
16
17
     int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype,
                   MPI_Count count)
18
19
20
     A.3.12 I/O C Bindings
21
22
     int MPI_CONVERSION_FN_NULL(void *userbuf, MPI_Datatype datatype, int count,
23
                   void *filebuf, MPI_Offset position, void *extra_state)
24
     int MPI_File_close(MPI_File *fh)
25
26
     int MPI_File_delete(const char *filename, MPI_Info info)
27
     int MPI_File_get_amode(MPI_File fh, int *amode)
28
29
     int MPI_File_get_atomicity(MPI_File fh, int *flag)
30
     int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,
^{31}
                   MPI_Offset *disp)
32
33
     int MPI_File_get_group(MPI_File fh, MPI_Group *group)
34
     int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
35
36
     int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
37
     int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)
38
39
     int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
40
     int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
41
42
                   MPI_Aint *extent)
43
     int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
44
                   MPI_Datatype *filetype, char *datarep)
45
46
     int MPI_File_iread_all(MPI_File fh, void *buf, int count,
47
                   MPI_Datatype datatype, MPI_Request *request)
48
```

int	<pre>MPI_File_iread_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>
int	<pre>MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>
int	MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
int	MPI_File_iread_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
int	<pre>MPI_File_iwrite_all(MPI_File fh, const void *buf, int count,</pre>
int	<pre>MPI_File_iwrite_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
int	<pre>MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
int	MPI_File_iwrite(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)12
int	<pre>MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count,</pre>
int	MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info, MPI_File *fh)2
int	MPI_File_preallocate(MPI_File fh, MPI_Offset size) 2 2 2
int	MPI_File_read_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)2
int	MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status) 3 3
int	MPI_File_read_all(MPI_File fh, void *buf, int count,3MPI_Datatype datatype, MPI_Status *status)3
int	<pre>MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,</pre>
int	MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status) ³
int	MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
int	<pre>MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>
int	MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
int	MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype) 4

ANNEX A. LANGUAGE BINDINGS SUMMARY

1 int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status) $\mathbf{2}$ int MPI_File_read_ordered(MPI_File fh, void *buf, int count, 3 MPI_Datatype datatype, MPI_Status *status) 4 $\mathbf{5}$ int MPI_File_read_shared(MPI_File fh, void *buf, int count, 6 MPI_Datatype datatype, MPI_Status *status) 7 int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence) 8 9 int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence) 10 int MPI_File_set_atomicity(MPI_File fh, int flag) 11 12int MPI_File_set_info(MPI_File fh, MPI_Info info) 13int MPI_File_set_size(MPI_File fh, MPI_Offset size) 1415int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, 16MPI_Datatype filetype, const char *datarep, MPI_Info info) 17int MPI_File_sync(MPI_File fh) 18 19int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count, 20MPI_Datatype datatype) 21int MPI_File_write_all_end(MPI_File fh, const void *buf, 22MPI_Status *status) 23 24 int MPI_File_write_all(MPI_File fh, const void *buf, int count, 25MPI_Datatype datatype, MPI_Status *status) 26int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, 27const void *buf, int count, MPI_Datatype datatype) 28 29int MPI_File_write_at_all_end(MPI_File fh, const void *buf, 30 MPI_Status *status) 31 32int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, 33 int count, MPI_Datatype datatype, MPI_Status *status) 34int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 35 int count, MPI_Datatype datatype, MPI_Status *status) 36 37 int MPI_File_write(MPI_File fh, const void *buf, int count, 38MPI_Datatype datatype, MPI_Status *status) 39

int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count, MPI_Datatype datatype)

⁴² int MPI_File_write_ordered_end(MPI_File fh, const void *buf, ⁴³ MPI_Status *status) ⁴⁴ ⁴⁵ int MPI_File_write_ordered(MPI_File fh, const void *buf, int count,

45 MPI_Datatype datatype, MPI_Status *status)

int MPI_File_write_shared(MPI_File fh, const void *buf, int count,

40

41

47

48

1 MPI_Datatype datatype, MPI_Status *status) $\mathbf{2}$ int MPI_Register_datarep(const char *datarep, 3 MPI_Datarep_conversion_function *read_conversion_fn, 4 MPI_Datarep_conversion_function *write_conversion_fn, 5 MPI_Datarep_extent_function *dtype_file_extent_fn, 6 void *extra_state) 7 8 9 A.3.13 Language Bindings C Bindings 10 int MPI_Status_f082f(const MPI_F08_status *f08_status, MPI_Fint *f_status) 11 12int MPI_Status_f2f08(const MPI_Fint *f_status, MPI_F08_status *f08_status) 13 int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype) 1415int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype) 16int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype) 1718 int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype) 19 MPI_Fint MPI_Comm_c2f(MPI_Comm comm) 2021MPI_Comm MPI_Comm_f2c(MPI_Fint comm) 22 MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler) 23 24 MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler) 2526MPI_Fint MPI_File_c2f(MPI_File file) 27MPI_File MPI_File_f2c(MPI_Fint file) 2829 MPI_Fint MPI_Group_c2f(MPI_Group group) 30 MPI_Group MPI_Group_f2c(MPI_Fint group) 3132 MPI_Fint MPI_Info_c2f(MPI_Info info) 33 MPI_Info MPI_Info_f2c(MPI_Fint info) 34 35 MPI_Fint MPI_Message_c2f(MPI_Message message) 36 MPI_Message MPI_Message_f2c(MPI_Fint message) 37 38 MPI_Fint MPI_Op_c2f(MPI_Op op) 39 MPI_Op MPI_Op_f2c(MPI_Fint op) 40 41 MPI_Fint MPI_Request_c2f(MPI_Request request) 42MPI_Request MPI_Request_f2c(MPI_Fint request) 43 44 MPI_Fint MPI_Session_c2f(MPI_Session session) 45MPI_Session MPI_Session_f2c(MPI_Fint session) 4647

48

```
1
     int MPI_Status_c2f08(const MPI_Status *c_status,
\mathbf{2}
                   MPI_F08_status *f08_status)
3
     int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)
4
\mathbf{5}
     int MPI_Status_f082c(const MPI_F08_status *f08_status,
6
                   MPI_Status *c_status)
7
     int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)
8
9
     MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
10
     MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
11
12
     MPI_Fint MPI_Win_c2f(MPI_Win win)
13
     MPI_Win MPI_Win_f2c(MPI_Fint win)
14
15
16
     A.3.14 Tools / Profiling Interface C Bindings
17
18
     int MPI_Pcontrol(const int level, ...)
19
20
     A.3.15 Tools / MPI Tool Information Interface C Bindings
21
22
     int MPI_T_category_changed(int *stamp)
23
     int MPI_T_category_get_categories(int cat_index, int len, int indices[])
^{24}
25
     int MPI_T_category_get_cvars(int cat_index, int len, int indices[])
26
     int MPI_T_category_get_events(int cat_index, int len, int indices[])
27
28
     int MPI_T_category_get_index(const char *name, int *cat_index)
29
     int MPI_T_category_get_info(int cat_index, char *name, int *name_len,
30
                   char *desc, int *desc_len, int *num_cvars, int *num_pvars,
31
                   int *num_categories)
32
33
     int MPI_T_category_get_num_events(int cat_index, int *num_events)
34
     int MPI_T_category_get_num(int *num_cat)
35
36
     int MPI_T_category_get_pvars(int cat_index, int len, int indices[])
37
38
     int MPI_T_cvar_get_index(const char *name, int *cvar_index)
39
     int MPI_T_cvar_get_info(int cvar_index, char *name, int *name_len,
40
                   int *verbosity, MPI_Datatype *datatype, MPI_T_enum *enumtype,
41
                   char *desc, int *desc_len, int *bind, int *scope)
42
43
     int MPI_T_cvar_get_num(int *num_cvar)
44
     int MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle,
45
                   MPI_T_cvar_handle *handle, int *count)
46
47
     int MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)
48
```

<pre>int MPI_T_cvar_read(MPI_T_cvar_handle handle, void *buf)</pre>	1
<pre>int MPI_T_cvar_write(MPI_T_cvar_handle handle, const void *buf)</pre>	2 3
<pre>int MPI_T_enum_get_info(MPI_T_enum enumtype, int *num, char *name,</pre>	4
int *name_len)	5
int MPI_T_enum_get_item(MPI_T_enum enumtype, int index, int *value,	6
char *name, int *name_len)	7 8
<pre>int MPI_T_event_callback_get_info(</pre>	9
MPI_T_event_registration event_registration,	10
MPI_T_cb_safety cb_safety, MPI_Info *info_used)	11
<pre>int MPI_T_event_callback_set_info(</pre>	12 13
MPI_T_event_registration event_registration,	14
<pre>MPI_T_cb_safety cb_safety, MPI_Info info)</pre>	15
<pre>int MPI_T_event_copy(MPI_T_event_instance event_instance, void *buffer)</pre>	16 17
<pre>int MPI_T_event_get_index(const char *name, int *event_index)</pre>	18
<pre>int MPI_T_event_get_info(int event_index, char *name, int *name_len,</pre>	19
int *verbosity, MPI_Datatype array_of_datatypes[],	20
MPI_Aint array_of_displacements[], int *num_elements,	21
<pre>MPI_T_enum *enumtype, MPI_Info *info, char *desc,</pre>	22 23
<pre>int *desc_len, int *bind)</pre>	23 24
<pre>int MPI_T_event_get_num(int *num_events)</pre>	25
<pre>int MPI_T_event_get_source(MPI_T_event_instance event_instance,</pre>	26
int *source_index)	27 28
<pre>int MPI_T_event_get_timestamp(MPI_T_event_instance event_instance,</pre>	29
MPI_Count *event_timestamp)	30
<pre>int MPI_T_event_handle_alloc(int event_index, void *obj_handle,</pre>	31
MPI_Info info, MPI_T_event_registration *event_registration)	32 33
int MPI_T_event_handle_free(MPI_T_event_registration event_registration,	34
void *user_data,	35
MPI_T_event_free_cb_function free_cb_function)	36
<pre>int MPI_T_event_handle_get_info(</pre>	37
MPI_T_event_registration event_registration,	38
MPI_Info *info_used)	39 40
<pre>int MPI_T_event_handle_set_info(</pre>	41
MPI_T_event_registration event_registration, MPI_Info info)	42
<pre>int MPI_T_event_read(MPI_T_event_instance event_instance,</pre>	43
int element_index, void *buffer)	44 45
<pre>int MPI_T_event_register_callback(</pre>	46
MPI_T_event_registration event_registration,	47
	48

	844	ANNEX A. LANGUAGE BINDINGS SUMMARY
1 2		<pre>MPI_T_cb_safety cb_safety, MPI_Info info, void *user_data, MPI_T_event_cb_function event_cb_function)</pre>
3 4 5 6	int	<pre>MPI_T_event_set_dropped_handler(MPI_T_event_registration event_registration, MPI_T_event_dropped_cb_function dropped_cb_function)</pre>
7	int	MPI_T_finalize(void)
8 9	int	MPI_T_init_thread(int required, int *provided)
10	int	MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)
11 12 13 14 15	int	<pre>MPI_T_pvar_get_info(int pvar_index, char *name, int *name_len,</pre>
16	int	MPI_T_pvar_get_num(int *num_pvar)
17 18 19	int	<pre>MPI_T_pvar_handle_alloc(MPI_T_pvar_session session, int pvar_index, void *obj_handle, MPI_T_pvar_handle *handle, int *count)</pre>
20 21 22	int	<pre>MPI_T_pvar_handle_free(MPI_T_pvar_session session,</pre>
22 23 24	int	<pre>MPI_T_pvar_read(MPI_T_pvar_session session, MPI_T_pvar_handle handle, void *buf)</pre>
25 26 27	int	<pre>MPI_T_pvar_readreset(MPI_T_pvar_session session,</pre>
28	int	<pre>MPI_T_pvar_reset(MPI_T_pvar_session session, MPI_T_pvar_handle handle)</pre>
29 30	int	MPI_T_pvar_session_create(MPI_T_pvar_session *session)
31	int	MPI_T_pvar_session_free(MPI_T_pvar_session *session)
32 33	int	MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle)
34	int	MPI_T_pvar_stop(MPI_T_pvar_session session, MPI_T_pvar_handle handle)
35 36 37	int	<pre>MPI_T_pvar_write(MPI_T_pvar_session session, MPI_T_pvar_handle handle,</pre>
38 39 40 41 42	int	<pre>MPI_T_source_get_info(int source_index, char *name, int *name_len,</pre>
42	int	MPI_T_source_get_num(int *num_sources)
44 45 46 47 48	int	<pre>MPI_T_source_get_timestamp(int source_index, MPI_Count *timestamp)</pre>

A.3.16 Deprecated C Bindings	1
<pre>int MPI_Attr_delete(MPI_Comm comm, int keyval)</pre>	2 3
<pre>int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)</pre>	4
int MPI_Attr_put(MPI_Comm comm, int keyval, void *attribute_val)	5
· · ·	6 7
<pre>int MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,</pre>	8
<pre>int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,</pre>	10 11
<pre>int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,</pre>	12 13
<pre>int MPI_Keyval_create(MPI_Copy_function *copy_fn,</pre>	14 15
void *extra_state)	16 17
<pre>int MPI_Keyval_free(int *keyval)</pre>	18 19
int MPI_NULL_COPY_FN(MPI_Comm oldcomm, int keyval, void *extra_state,	20
<pre>void *attribute_val_in, void *attribute_val_out, int *flag)</pre>	21
<pre>int MPI_NULL_DELETE_FN(MPI_Comm comm, int keyval, void *attribute_val,</pre>	22 23
void *extra_state)	24
	25
	26
	27 28
	29
	30
	31
	32
	33 34
	35
	36
	37
	38 39
	40
	41
	42
	43
	44 45
	46
	47
	48

```
A.4 Fortran 2008 Bindings with the mpi_f08 Module
1
\mathbf{2}
     A.4.1 Point-to-Point Communication Fortran 2008 Bindings
3
4
     MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)
5
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
6
         INTEGER, INTENT(IN) :: count, dest, tag
7
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
12
         INTEGER, INTENT(IN) :: count, dest, tag
13
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
         TYPE(MPI_Comm), INTENT(IN) :: comm
15
         TYPE(MPI_Request), INTENT(OUT) :: request
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Buffer_attach(buffer, size, ierror) <
19
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
20
         INTEGER, INTENT(IN) :: size
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_Buffer_detach(buffer_addr, size, ierror)
24
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
25
         TYPE(C_PTR), INTENT(OUT) :: buffer_addr
26
         INTEGER, INTENT(OUT) :: size
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Cancel(request, ierror)
29
         TYPE(MPI_Request), INTENT(IN) :: request
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_Get_count(status, datatype, count, ierror)
33
         TYPE(MPI_Status), INTENT(IN) :: status
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
         INTEGER, INTENT(OUT) :: count
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
     MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror)
38
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
39
         INTEGER, INTENT(IN) :: count, dest, tag
40
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         TYPE(MPI_Request), INTENT(OUT) :: request
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_Improbe(source, tag, comm, flag, message, status, ierror)
46
         INTEGER, INTENT(IN) :: source, tag
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

```
1
    LOGICAL, INTENT(OUT) :: flag
                                                                                   2
    TYPE(MPI_Message), INTENT(OUT) :: message
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   5
MPI_Imrecv(buf, count, datatype, message, request, ierror)
                                                                                   6
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   9
    TYPE(MPI_Message), INTENT(INOUT) :: message
                                                                                  10
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  12
                                                                                  13
MPI_Iprobe(source, tag, comm, flag, status, ierror)
                                                                                  14
    INTEGER, INTENT(IN) :: source, tag
                                                                                  15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  16
    LOGICAL, INTENT(OUT) :: flag
                                                                                  17
    TYPE(MPI_Status) :: status
                                                                                  18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  19
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
                                                                                  20
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  21
    INTEGER, INTENT(IN) :: count, source, tag
                                                                                  22
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  23
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  24
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  26
                                                                                  27
MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)
                                                                                  28
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  29
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                  30
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  31
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  32
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  34
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
                                                                                  35
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  36
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                  37
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  38
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  39
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  41
                                                                                  42
MPI_Isendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
                                                                                  43
             comm, request, ierror)
                                                                                  44
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  45
    INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
                                                                                  46
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  47
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  48
```

```
1
         TYPE(MPI_Request), INTENT(OUT) :: request
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
4
                   recvcount, recvtype, source, recvtag, comm, request, ierror)
5
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
6
         INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
7
                   recvtag
8
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
9
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)
15
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
16
         INTEGER, INTENT(IN) :: count, dest, tag
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         TYPE(MPI_Comm), INTENT(IN) :: comm
19
         TYPE(MPI_Request), INTENT(OUT) :: request
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_Mprobe(source, tag, comm, message, status, ierror)
22
         INTEGER, INTENT(IN) :: source, tag
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
24
         TYPE(MPI_Message), INTENT(OUT) :: message
25
         TYPE(MPI_Status) :: status
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Mrecv(buf, count, datatype, message, status, ierror)
29
         TYPE(*), DIMENSION(..) :: buf
30
         INTEGER, INTENT(IN) :: count
31
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
         TYPE(MPI_Message), INTENT(INOUT) :: message
33
         TYPE(MPI_Status) :: status
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_Probe(source, tag, comm, status, ierror)
36
         INTEGER, INTENT(IN) :: source, tag
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         TYPE(MPI_Status) :: status
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
42
         TYPE(*), DIMENSION(..) :: buf
43
         INTEGER, INTENT(IN) :: count, source, tag
44
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
45
         TYPE(MPI_Comm), INTENT(IN) :: comm
46
         TYPE(MPI_Status) :: status
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)	1
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	2
INTEGER, INTENT(IN) :: count, source, tag	3
TYPE(MPI_Datatype), INTENT(IN) :: datatype	4
TYPE(MPI_Comm), INTENT(IN) :: comm	5
TYPE(MPI_Request), INTENT(OUT) :: request	6 7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8
MPI_Request_free(request, ierror)	9
TYPE(MPI_Request), INTENT(INOUT) :: request	10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	11
MPI_Request_get_status(request, flag, status, ierror)	12
TYPE(MPI_Request), INTENT(IN) :: request	13
LOGICAL, INTENT(OUT) :: flag	14
TYPE(MPI_Status) :: status	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	16
	17
MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)	18
TYPE(*), DIMENSION(), INTENT(IN) :: buf	19
INTEGER, INTENT(IN) :: count, dest, tag	20
TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm	21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	22
INTEGER, OFFICIARE, INTENT(OOF) TETTOT	23
MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)	24
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	25 26
INTEGER, INTENT(IN) :: count, dest, tag	20 27
TYPE(MPI_Datatype), INTENT(IN) :: datatype	21
TYPE(MPI_Comm), INTENT(IN) :: comm	20
TYPE(MPI_Request), INTENT(OUT) :: request	30
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	31
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)	32
<pre>TYPE(*), DIMENSION(), INTENT(IN) :: buf</pre>	33
INTEGER, INTENT(IN) :: count, dest, tag	34
TYPE(MPI_Datatype), INTENT(IN) :: datatype	35
TYPE(MPI_Comm), INTENT(IN) :: comm	36
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	37
MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)	38
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	39
INTEGER, INTENT(IN) :: count, dest, tag	40
TYPE(MPI_Datatype), INTENT(IN) :: datatype	41
TYPE(MPI_Comm), INTENT(IN) :: comm	42
TYPE(MPI_Request), INTENT(OUT) :: request	43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,	45 46
comm, status, ierror)	40 47
TYPE(*), DIMENSION() :: buf	48

```
1
         INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
2
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         TYPE(MPI_Status) :: status
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
7
                   recvcount, recvtype, source, recvtag, comm, status, ierror)
8
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
9
         INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
10
                   recvtag
11
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
12
         TYPE(*), DIMENSION(..) :: recvbuf
13
         TYPE(MPI_Comm), INTENT(IN) :: comm
14
         TYPE(MPI_Status) :: status
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)
18
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
19
         INTEGER, INTENT(IN) :: count, dest, tag
20
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
24
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
25
         INTEGER, INTENT(IN) :: count, dest, tag
26
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
         TYPE(MPI_Comm), INTENT(IN) :: comm
28
         TYPE(MPI_Request), INTENT(OUT) :: request
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
     MPI_Startall(count, array_of_requests, ierror)
32
         INTEGER, INTENT(IN) :: count
33
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_Start(request, ierror)
36
         TYPE(MPI_Request), INTENT(INOUT) :: request
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
40
         INTEGER, INTENT(IN) :: count
41
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
42
         LOGICAL, INTENT(OUT) :: flag
43
         TYPE(MPI_Status) :: array_of_statuses(*)
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_Testany(count, array_of_requests, index, flag, status, ierror)
46
         INTEGER, INTENT(IN) :: count
47
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
48
```

```
1
    INTEGER, INTENT(OUT) :: index
                                                                                   2
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   5
MPI_Test_cancelled(status, flag, ierror)
                                                                                   6
    TYPE(MPI_Status), INTENT(IN) :: status
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  10
MPI_Test(request, flag, status, ierror)
                                                                                  11
    TYPE(MPI_Request), INTENT(INOUT) :: request
    LOGICAL, INTENT(OUT) :: flag
                                                                                  12
                                                                                  13
    TYPE(MPI_Status) :: status
                                                                                  14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  15
MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
                                                                                  16
             array_of_statuses, ierror)
                                                                                  17
    INTEGER, INTENT(IN) :: incount
                                                                                  18
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
                                                                                  19
    INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
                                                                                  20
    TYPE(MPI_Status) :: array_of_statuses(*)
                                                                                  21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  22
                                                                                  23
MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
                                                                                  24
    INTEGER, INTENT(IN) :: count
                                                                                  25
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                  26
    TYPE(MPI_Status) :: array_of_statuses(*)
                                                                                  27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  28
MPI_Waitany(count, array_of_requests, index, status, ierror)
                                                                                  29
    INTEGER, INTENT(IN) :: count
                                                                                  30
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                  31
    INTEGER, INTENT(OUT) :: index
                                                                                  32
    TYPE(MPI_Status) :: status
                                                                                  33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  34
                                                                                  35
MPI_Wait(request, status, ierror)
                                                                                  36
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                  37
    TYPE(MPI_Status) :: status
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
                                                                                  40
             array_of_statuses, ierror)
                                                                                  41
    INTEGER, INTENT(IN) :: incount
                                                                                  42
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
                                                                                  43
    INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
                                                                                  44
    TYPE(MPI_Status) :: array_of_statuses(*)
                                                                                  45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  46
                                                                                  47
                                                                                  48
```

```
1
     A.4.2 Partitioned Communication Fortran 2008 Bindings
\mathbf{2}
     MPI_Parrived(request, partition, flag, ierror)
3
         TYPE(MPI_Request), INTENT(INOUT) :: request
4
         INTEGER, INTENT(IN) :: partition
5
         LOGICAL, INTENT(OUT) :: flag
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Pready_list(length, array_of_partitions, request, ierror)
9
         INTEGER, INTENT(IN) :: length
10
         INTEGER, INTENT(INOUT) :: array_of_partitions(length)
11
         TYPE(MPI_Request), INTENT(INOUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_Pready(partition, request, ierror)
14
         INTEGER, INTENT(IN) :: partition
15
         TYPE(MPI_Request), INTENT(INOUT) :: request
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Pready_range(partition_low, partition_high, request, ierror)
19
         INTEGER, INTENT(IN) :: partition_low, partition_high
20
         TYPE(MPI_Request), INTENT(INOUT) :: request
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     MPI_Precv_init(buf, partitions, count, datatype, dest, tag, comm, info,
23
                   request, ierror)
24
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
25
         INTEGER, INTENT(IN) :: partitions, dest, tag
26
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
27
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
         TYPE(MPI_Comm), INTENT(IN) :: comm
29
         TYPE(MPI_Info), INTENT(IN) :: info
30
         TYPE(MPI_Request), INTENT(OUT) :: request
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Psend_init(buf, partitions, count, datatype, dest, tag, comm, info,
34
                   request, ierror)
35
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
36
         INTEGER, INTENT(IN) :: partitions, dest, tag
37
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         TYPE(MPI_Info), INTENT(IN) :: info
41
         TYPE(MPI_Request), INTENT(OUT) :: request
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     A.4.3 Datatypes Fortran 2008 Bindings
45
46
     INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_add(base, disp)
47
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: base, disp
48
```

INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_diff(addr1, addr2) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: addr1, addr2	1 2
	3
MPI_Get_address(location, address, ierror)	4
TYPE(*), DIMENSION(), ASYNCHRONOUS :: location	5
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address	6
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	7
MPI_Get_elements(status, datatype, count, ierror)	8
TYPE(MPI_Status), INTENT(IN) :: status	9
TYPE(MPI_Datatype), INTENT(IN) :: datatype	10
INTEGER, INTENT(OUT) :: count	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12
	13
MPI_Get_elements_x(status, datatype, count, ierror)	14
TYPE(MPI_Status), INTENT(IN) :: status	15
TYPE(MPI_Datatype), INTENT(IN) :: datatype	16
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count	17
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	18
	19
MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,	20
position, ierror)	
CHARACTER(LEN=*), INTENT(IN) :: datarep	21
TYPE(*), DIMENSION(), INTENT(IN) :: inbuf	22
INTEGER, INTENT(IN) :: incount	23
TYPE(MPI_Datatype), INTENT(IN) :: datatype	24
TYPE(*), DIMENSION() :: outbuf	25
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize	26
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position	27
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	28
	29
MPI_Pack_external_size(datarep, incount, datatype, size, ierror)	30
CHARACTER(LEN=*), INTENT(IN) :: datarep	31
INTEGER, INTENT(IN) :: incount	32
TYPE(MPI_Datatype), INTENT(IN) :: datatype	33
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size	34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
	35
MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)	36
TYPE(*), DIMENSION(), INTENT(IN) :: inbuf	37
INTEGER, INTENT(IN) :: incount, outsize	38
TYPE(MPI_Datatype), INTENT(IN) :: datatype	39
TYPE(*), DIMENSION() :: outbuf	40
INTEGER, INTENT(INOUT) :: position	41
TYPE(MPI_Comm), INTENT(IN) :: comm	42
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	43
INIEGEN, OFIIONAE, INIENI(UUI) ICIIUI	44
MPI_Pack_size(incount, datatype, comm, size, ierror)	45
INTEGER, INTENT(IN) :: incount	46
TYPE(MPI_Datatype), INTENT(IN) :: datatype	47
TYPE(MPI_Comm), INTENT(IN) :: comm	48
	-10

```
1
         INTEGER, INTENT(OUT) :: size
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Type_commit(datatype, ierror)
4
         TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7
     MPI_Type_contiguous(count, oldtype, newtype, ierror)
8
         INTEGER, INTENT(IN) :: count
9
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
10
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
13
                   array_of_distribs, array_of_dargs, array_of_psizes, order,
14
                   oldtype, newtype, ierror)
15
         INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
16
                   array_of_distribs(ndims), array_of_dargs(ndims),
17
                   array_of_psizes(ndims), order
18
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
19
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
23
                   oldtype, newtype, ierror)
24
         INTEGER, INTENT(IN) :: count, blocklength
25
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN)
                                                      1:
26
                   array_of_displacements(count)
27
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
28
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
     MPI_Type_create_hindexed(count, array_of_blocklengths,
31
                   array_of_displacements, oldtype, newtype, ierror)
32
         INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
33
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
34
                   array_of_displacements(count)
35
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
36
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
40
                   ierror)
41
         INTEGER, INTENT(IN) :: count, blocklength
42
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
43
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
44
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
     MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
47
                   oldtype, newtype, ierror)
48
```

```
1
    INTEGER, INTENT(IN) :: count, blocklength,
                                                                                  2
              array_of_displacements(count)
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  5
                                                                                  6
MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror)
                                                                                  7
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent
                                                                                  9
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  11
MPI_Type_create_struct(count, array_of_blocklengths,
                                                                                  12
                                                                                  13
              array_of_displacements, array_of_types, newtype, ierror)
                                                                                  14
    INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
                                                                                  15
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
                                                                                  16
              array_of_displacements(count)
                                                                                  17
    TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
                                                                                  18
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  20
MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
                                                                                  21
              array_of_starts, order, oldtype, newtype, ierror)
                                                                                  22
    INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),
                                                                                  23
              array_of_subsizes(ndims), array_of_starts(ndims), order
                                                                                  24
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                  25
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  27
                                                                                  28
MPI_Type_dup(oldtype, newtype, ierror)
                                                                                  29
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                  30
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  32
MPI_Type_free(datatype, ierror)
                                                                                  33
    TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
                                                                                  34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  35
                                                                                  36
MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,
                                                                                  37
              array_of_integers, array_of_addresses, array_of_datatypes,
                                                                                  38
              ierror)
                                                                                  39
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  40
    INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes
                                                                                  41
    INTEGER, INTENT(OUT) :: array_of_integers(max_integers)
                                                                                  42
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
                                                                                  43
              array_of_addresses(max_addresses)
                                                                                  44
    TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)
                                                                                  45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  46
MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,
                                                                                  47
              combiner, ierror)
                                                                                  48
```

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
2
         INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,
3
                   combiner
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Type_get_extent(datatype, lb, extent, ierror)
6
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
7
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
     MPI_Type_get_extent_x(datatype, lb, extent, ierror)
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: lb, extent
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
15
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
16
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror)
20
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
21
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
24
                   oldtype, newtype, ierror)
25
         INTEGER, INTENT(IN) :: count, array_of_blocklengths(count),
26
                   array_of_displacements(count)
27
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
28
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
     MPI_Type_size(datatype, size, ierror)
32
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
33
         INTEGER, INTENT(OUT) :: size
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_Type_size_x(datatype, size, ierror)
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
41
         INTEGER, INTENT(IN) :: count, blocklength, stride
42
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
43
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
46
                   datatype, ierror)
47
         CHARACTER(LEN=*), INTENT(IN) :: datarep
48
```

```
TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                   1
                                                                                   2
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
    TYPE(*), DIMENSION(...) :: outbuf
    INTEGER, INTENT(IN) :: outcount
                                                                                   5
                                                                                   6
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
              ierror)
                                                                                   10
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                   11
    INTEGER, INTENT(IN) :: insize, outcount
                                                                                   12
    INTEGER, INTENT(INOUT) :: position
                                                                                   13
    TYPE(*), DIMENSION(..) :: outbuf
                                                                                   14
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   17
                                                                                   18
                                                                                   19
A.4.4 Collective Communication Fortran 2008 Bindings
                                                                                   20
MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  21
             recvtype, comm, info, request, ierror)
                                                                                  22
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  23
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  24
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  25
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   26
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  27
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  28
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  30
                                                                                   31
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                   32
             comm, ierror)
                                                                                  33
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  34
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  35
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  36
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  37
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                   40
             displs, recvtype, comm, info, request, ierror)
                                                                                  41
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  42
    INTEGER, INTENT(IN) :: sendcount
                                                                                  43
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  44
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   45
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                   46
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   47
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   48
```

```
1
         TYPE(MPI_Request), INTENT(OUT) :: request
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
4
                  recvtype, comm, ierror)
5
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
6
         INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
7
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
8
         TYPE(*), DIMENSION(..) :: recvbuf
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
    MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
13
                  request, ierror)
14
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
15
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
16
         INTEGER, INTENT(IN) :: count
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         TYPE(MPI_Op), INTENT(IN) :: op
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Info), INTENT(IN) :: info
21
         TYPE(MPI_Request), INTENT(OUT) :: request
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
24
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
25
         TYPE(*), DIMENSION(..) :: recvbuf
26
         INTEGER, INTENT(IN) :: count
27
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
         TYPE(MPI_Op), INTENT(IN) :: op
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
33
                  recvtype, comm, info, request, ierror)
34
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
35
         INTEGER, INTENT(IN) :: sendcount, recvcount
36
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
37
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         TYPE(MPI_Info), INTENT(IN) :: info
40
         TYPE(MPI_Request), INTENT(OUT) :: request
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
43
                  comm, ierror)
44
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
45
         INTEGER, INTENT(IN) :: sendcount, recvcount
46
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
47
         TYPE(*), DIMENSION(..) :: recvbuf
48
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   1
                                                                                   2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
              recvcounts, rdispls, recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   6
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   9
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   10
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   11
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   12
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   14
                                                                                   15
MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
                                                                                   16
              rdispls, recvtype, comm, ierror)
                                                                                   17
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                   18
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
                                                                                   19
              rdispls(*)
                                                                                  20
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  21
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                  22
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   24
MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                  25
              recvcounts, rdispls, recvtypes, comm, info, request, ierror)
                                                                                   26
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  27
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                  28
              recvcounts(*), rdispls(*)
                                                                                  29
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                  30
              recvtypes(*)
                                                                                   31
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                   32
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   33
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  34
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  36
                                                                                  37
MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
                                                                                  38
             rdispls, recvtypes, comm, ierror)
                                                                                  39
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                   40
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
                                                                                  41
              rdispls(*)
                                                                                  42
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
                                                                                   43
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                   44
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   46
MPI_Barrier(comm, ierror)
                                                                                   47
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Barrier_init(comm, info, request, ierror)
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         TYPE(MPI_Info), INTENT(IN) :: info
5
         TYPE(MPI_Request), INTENT(OUT) :: request
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Bcast(buffer, count, datatype, root, comm, ierror)
9
         TYPE(*), DIMENSION(..) :: buffer
10
         INTEGER, INTENT(IN) :: count, root
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
15
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
16
         INTEGER, INTENT(IN) :: count, root
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         TYPE(MPI_Comm), INTENT(IN) :: comm
19
         TYPE(MPI_Info), INTENT(IN) :: info
20
         TYPE(MPI_Request), INTENT(OUT) :: request
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
24
                   ierror)
25
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
26
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
27
         INTEGER, INTENT(IN) :: count
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Op), INTENT(IN) :: op
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         TYPE(MPI_Info), INTENT(IN) :: info
32
         TYPE(MPI_Request), INTENT(OUT) :: request
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
35
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
36
         TYPE(*), DIMENSION(...) :: recvbuf
37
         INTEGER, INTENT(IN) :: count
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         TYPE(MPI_Op), INTENT(IN) :: op
40
         TYPE(MPI_Comm), INTENT(IN) :: comm
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
44
                   root, comm, info, request, ierror)
45
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
46
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
47
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
48
```

```
TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  1
                                                                                  2
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                  10
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  11
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                  12
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  14
                                                                                  15
MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  16
              recvtype, root, comm, info, request, ierror)
                                                                                  17
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  18
    INTEGER, INTENT(IN) :: sendcount, root
                                                                                  19
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  20
                                                                                  21
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                  22
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  23
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  24
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  26
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  27
             recvtype, root, comm, ierror)
                                                                                  28
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  29
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
                                                                                  30
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  31
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  32
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  34
                                                                                  35
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  36
              comm, request, ierror)
                                                                                  37
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  38
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  39
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  40
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  41
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  42
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  44
MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  45
              recvtype, comm, request, ierror)
                                                                                  46
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  47
    INTEGER, INTENT(IN) :: sendcount
                                                                                  48
```

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
2
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
3
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
4
         TYPE(MPI_Comm), INTENT(IN) :: comm
5
         TYPE(MPI_Request), INTENT(OUT) :: request
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
8
                  ierror)
9
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
10
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
11
         INTEGER, INTENT(IN) :: count
12
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
13
         TYPE(MPI_Op), INTENT(IN) :: op
14
         TYPE(MPI_Comm), INTENT(IN) :: comm
15
         TYPE(MPI_Request), INTENT(OUT) :: request
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
19
                  comm, request, ierror)
20
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
21
         INTEGER, INTENT(IN) :: sendcount, recvcount
22
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
23
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
24
         TYPE(MPI_Comm), INTENT(IN) :: comm
25
         TYPE(MPI_Request), INTENT(OUT) :: request
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
28
                  rdispls, recvtype, comm, request, ierror)
29
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
30
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
31
               recvcounts(*), rdispls(*)
32
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
33
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
         TYPE(MPI_Request), INTENT(OUT) :: request
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
39
                  recvcounts, rdispls, recvtypes, comm, request, ierror)
40
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
41
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
42
                   recvcounts(*), rdispls(*)
43
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
44
                   recvtypes(*)
45
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
46
         TYPE(MPI_Comm), INTENT(IN) :: comm
47
         TYPE(MPI_Request), INTENT(OUT) :: request
48
```

```
1
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   2
MPI_Ibarrier(comm, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: count, root
                                                                                   10
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   11
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   12
                                                                                   13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   14
MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
                                                                                   15
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   16
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   17
    INTEGER, INTENT(IN) :: count
                                                                                   18
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   19
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                   20
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  21
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  23
                                                                                   ^{24}
MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                   25
             root, comm, request, ierror)
                                                                                   26
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  27
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                  28
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  29
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  30
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   31
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  33
MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  34
             recvtype, root, comm, request, ierror)
                                                                                  35
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  36
    INTEGER, INTENT(IN) :: sendcount, displs(*), root
                                                                                  37
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  38
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   39
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                   40
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   41
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  43
                                                                                  44
MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                  45
             request, ierror)
                                                                                   46
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   47
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   48
```

```
1
         INTEGER, INTENT(IN) :: recvcount
2
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         TYPE(MPI_Op), INTENT(IN) :: op
4
         TYPE(MPI_Comm), INTENT(IN) :: comm
5
         TYPE(MPI_Request), INTENT(OUT) :: request
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
8
                  request, ierror)
9
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
10
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
11
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
12
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
13
         TYPE(MPI_Op), INTENT(IN) :: op
14
         TYPE(MPI_Comm), INTENT(IN) :: comm
15
         TYPE(MPI_Request), INTENT(OUT) :: request
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
19
                   ierror)
20
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
21
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
22
         INTEGER, INTENT(IN) :: count, root
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         TYPE(MPI_Op), INTENT(IN) :: op
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         TYPE(MPI_Request), INTENT(OUT) :: request
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
29
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
30
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
31
         INTEGER, INTENT(IN) :: count
32
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
33
         TYPE(MPI_Op), INTENT(IN) :: op
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
         TYPE(MPI_Request), INTENT(OUT) :: request
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
39
                  root, comm, request, ierror)
40
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
41
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
42
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
43
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
44
         TYPE(MPI_Comm), INTENT(IN) :: comm
45
         TYPE(MPI_Request), INTENT(OUT) :: request
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

```
1
MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
                                                                                   2
              recvtype, root, comm, request, ierror)
                                                                                   3
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*)
    INTEGER, INTENT(IN) :: displs(*), recvcount, root
                                                                                   5
                                                                                   6
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   11
MPI_Op_commutative(op, commute, ierror)
                                                                                   12
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                   13
    LOGICAL, INTENT(OUT) :: commute
                                                                                   14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   15
                                                                                   16
MPI_Op_create(user_fn, commute, op, ierror)
                                                                                   17
    PROCEDURE(MPI_User_function) :: user_fn
                                                                                   18
    LOGICAL, INTENT(IN) :: commute
                                                                                   19
    TYPE(MPI_Op), INTENT(OUT) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   20
                                                                                   21
MPI_Op_free(op, ierror)
                                                                                   22
    TYPE(MPI_Op), INTENT(INOUT) ::
                                    OD
                                                                                   23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   ^{24}
                                                                                   25
MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
                                                                                   26
              request, ierror)
                                                                                   27
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   28
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   29
    INTEGER, INTENT(IN) :: count, root
                                                                                   30
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   31
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                   32
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   33
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   34
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   36
MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
                                                                                   37
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                   38
    TYPE(*), DIMENSION(..) :: inoutbuf
                                                                                   39
    INTEGER, INTENT(IN) :: count
                                                                                   40
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   41
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                   42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   43
                                                                                   44
MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
                                                                                   45
              comm, info, request, ierror)
                                                                                   46
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   47
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   48
```

```
1
         INTEGER, INTENT(IN) :: recvcount
2
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         TYPE(MPI_Op), INTENT(IN) :: op
4
         TYPE(MPI_Comm), INTENT(IN) :: comm
5
         TYPE(MPI_Info), INTENT(IN) :: info
6
         TYPE(MPI_Request), INTENT(OUT) :: request
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
9
                   ierror)
10
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
11
         TYPE(*), DIMENSION(...) :: recvbuf
12
         INTEGER, INTENT(IN) :: recvcount
13
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
         TYPE(MPI_Op), INTENT(IN) :: op
15
         TYPE(MPI_Comm), INTENT(IN) :: comm
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
19
                   info, request, ierror)
20
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
21
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
22
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         TYPE(MPI_Op), INTENT(IN) :: op
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         TYPE(MPI_Info), INTENT(IN) :: info
27
         TYPE(MPI_Request), INTENT(OUT) :: request
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
30
                   ierror)
31
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
32
         TYPE(*), DIMENSION(..) :: recvbuf
33
         INTEGER, INTENT(IN) :: recvcounts(*)
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
         TYPE(MPI_Op), INTENT(IN) :: op
36
         TYPE(MPI_Comm), INTENT(IN) :: comm
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
40
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
41
         TYPE(*), DIMENSION(...) :: recvbuf
42
         INTEGER, INTENT(IN) :: count, root
43
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
44
         TYPE(MPI_Op), INTENT(IN) :: op
45
         TYPE(MPI_Comm), INTENT(IN) :: comm
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

```
MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
                                                                                   1
                                                                                   2
              ierror)
                                                                                   3
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
                                                                                   5
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   6
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  10
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  11
                                                                                  12
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
                                                                                  13
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  14
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                  15
    INTEGER, INTENT(IN) :: count
                                                                                  16
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  17
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  18
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  20
MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  21
                                                                                  22
             recvtype, root, comm, info, request, ierror)
                                                                                  23
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  24
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                  25
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  26
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  27
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  28
                                                                                  29
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  30
                                                                                  31
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  32
             root, comm, ierror)
                                                                                  33
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  34
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                  35
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  36
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  37
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
                                                                                  40
MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
                                                                                  41
             recvcount, recvtype, root, comm, info, request, ierror)
                                                                                  42
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  43
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
                                                                                  44
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  45
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  46
    INTEGER, INTENT(IN) :: recvcount, root
                                                                                  47
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  48
```

```
1
         TYPE(MPI_Info), INTENT(IN) :: info
2
         TYPE(MPI_Request), INTENT(OUT) :: request
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
5
                   recvtype, root, comm, ierror)
6
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
7
         INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
8
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
9
         TYPE(*), DIMENSION(..) :: recvbuf
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
14
     A.4.5 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings
15
    MPI_Comm_compare(comm1, comm2, result, ierror)
16
         TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
17
         INTEGER, INTENT(OUT) :: result
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     MPI_Comm_create(comm, group, newcomm, ierror)
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         TYPE(MPI_Group), INTENT(IN) :: group
23
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     MPI_Comm_create_from_group(group, stringtag, info, errhandler, newcomm,
                   ierror)
27
         TYPE(MPI_Group), INTENT(IN) :: group
28
         CHARACTER(LEN=*), INTENT(IN) :: stringtag
29
         TYPE(MPI_Info), INTENT(IN) :: info
30
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
31
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Comm_create_group(comm, group, tag, newcomm, ierror)
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
         TYPE(MPI_Group), INTENT(IN) :: group
37
         INTEGER, INTENT(IN) :: tag
38
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
41
42
                   extra_state, ierror)
         PROCEDURE (MPI_Comm_copy_attr_function), INTENT(IN) :: comm_copy_attr_fn
43
         PROCEDURE(MPI_Comm_delete_attr_function), INTENT(IN) ::
44
                   comm_delete_attr_fn
45
         INTEGER, INTENT(OUT) :: comm_keyval
46
47
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

1 MPI_Comm_delete_attr(comm, comm_keyval, ierror) 2 TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: comm_keyval INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Comm_dup(comm, newcomm, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(OUT) :: newcomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10 MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in, 11 attribute_val_out, flag, ierror) TYPE(MPI_Comm) :: oldcomm 1213 INTEGER :: comm_keyval, ierror 14INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in, 15attribute_val_out 16LOGICAL :: flag 17 MPI_Comm_dup_with_info(comm, info, newcomm, ierror) 18 TYPE(MPI_Comm), INTENT(IN) :: comm 19 TYPE(MPI_Info), INTENT(IN) :: info 20TYPE(MPI_Comm), INTENT(OUT) :: newcomm 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 23MPI_Comm_free(comm, ierror) 24TYPE(MPI_Comm), INTENT(INOUT) :: comm 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 26MPI_Comm_free_keyval(comm_keyval, ierror) 27INTEGER, INTENT(INOUT) :: comm_keyval 28INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 30 MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror) 31TYPE(MPI_Comm), INTENT(IN) :: comm 32 INTEGER, INTENT(IN) :: comm_keyval 33 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val 34 LOGICAL, INTENT(OUT) :: flag 35INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 MPI_Comm_get_info(comm, info_used, ierror) 37 TYPE(MPI_Comm), INTENT(IN) :: comm 38 TYPE(MPI_Info), INTENT(OUT) :: info_used 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 41 MPI_Comm_get_name(comm, comm_name, resultlen, ierror) 42TYPE(MPI_Comm), INTENT(IN) :: comm 43 CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name 44 INTEGER, INTENT(OUT) :: resultlen 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 46MPI_Comm_group(comm, group, ierror) 47TYPE(MPI_Comm), INTENT(IN) :: comm 48

```
1
         TYPE(MPI_Group), INTENT(OUT) :: group
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Comm_idup(comm, newcomm, request, ierror)
4
         TYPE(MPI_Comm), INTENT(IN) :: comm
5
         TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
6
         TYPE(MPI_Request), INTENT(OUT) :: request
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
     MPI_Comm_idup_with_info(comm, info, newcomm, request, ierror)
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         TYPE(MPI_Info), INTENT(IN) :: info
12
         TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
13
         TYPE(MPI_Request), INTENT(OUT) :: request
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
     MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
16
                   attribute_val_out, flag, ierror)
17
         TYPE(MPI_Comm) :: oldcomm
18
         INTEGER :: comm_keyval, ierror
19
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
20
                   attribute_val_out
21
         LOGICAL :: flag
22
23
     MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,
24
                   ierror)
25
         TYPE(MPI_Comm) :: comm
26
         INTEGER :: comm_keyval, ierror
27
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
28
     MPI_Comm_rank(comm, rank, ierror)
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         INTEGER, INTENT(OUT) :: rank
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Comm_remote_group(comm, group, ierror)
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
         TYPE(MPI_Group), INTENT(OUT) :: group
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
     MPI_Comm_remote_size(comm, size, ierror)
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         INTEGER, INTENT(OUT) :: size
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         INTEGER, INTENT(IN) :: comm_keyval
45
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
     MPI_Comm_set_info(comm, info, ierror)
48
```

TYPE(MPI_Comm), INTENT(IN) :: comm 1 TYPE(MPI_Info), INTENT(IN) :: info 2 INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Comm_set_name(comm, comm_name, ierror) 5 TYPE(MPI_Comm), INTENT(IN) :: comm 6 CHARACTER(LEN=*). INTENT(IN) :: comm name INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Comm_size(comm, size, ierror) 10 TYPE(MPI_Comm), INTENT(IN) :: comm 11 INTEGER, INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1213 MPI_Comm_split(comm, color, key, newcomm, ierror) 14 TYPE(MPI_Comm), INTENT(IN) :: comm 15INTEGER, INTENT(IN) :: color, key 16TYPE(MPI_Comm), INTENT(OUT) :: newcomm 17 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 19 MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror) 20TYPE(MPI_Comm), INTENT(IN) :: comm 21INTEGER, INTENT(IN) :: split_type, key 22 TYPE(MPI_Info), INTENT(IN) :: info 23TYPE(MPI_Comm), INTENT(OUT) :: newcomm 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 25MPI_Comm_test_inter(comm, flag, ierror) 26TYPE(MPI_Comm), INTENT(IN) :: comm 27LOGICAL, INTENT(OUT) :: flag 28 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 30 MPI_Group_compare(group1, group2, result, ierror) 31TYPE(MPI_Group), INTENT(IN) :: group1, group2 32 INTEGER, INTENT(OUT) :: result 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 MPI_Group_difference(group1, group2, newgroup, ierror) 35TYPE(MPI_Group), INTENT(IN) :: group1, group2 36 TYPE(MPI_Group), INTENT(OUT) :: newgroup 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 MPI_Group_excl(group, n, ranks, newgroup, ierror) 40 TYPE(MPI_Group), INTENT(IN) :: group 41 INTEGER, INTENT(IN) :: n, ranks(n) 42TYPE(MPI_Group), INTENT(OUT) :: newgroup 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44 MPI_Group_free(group, ierror) 45TYPE(MPI_Group), INTENT(INOUT) :: group 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4748

```
1
     MPI_Group_from_session_pset(session, pset_name, newgroup, ierror)
\mathbf{2}
         TYPE(MPI_Session), INTENT(IN) :: session
3
         CHARACTER(LEN=*), INTENT(IN) :: pset_name
4
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Group_incl(group, n, ranks, newgroup, ierror)
7
         TYPE(MPI_Group), INTENT(IN) :: group
8
         INTEGER, INTENT(IN) :: n, ranks(n)
9
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     MPI_Group_intersection(group1, group2, newgroup, ierror)
13
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
14
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
     MPI_Group_range_excl(group, n, ranges, newgroup, ierror)
17
         TYPE(MPI_Group), INTENT(IN) :: group
18
         INTEGER, INTENT(IN) :: n, ranges(3, n)
19
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
23
         TYPE(MPI_Group), INTENT(IN) :: group
24
         INTEGER, INTENT(IN) :: n, ranges(3, n)
25
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Group_rank(group, rank, ierror)
28
         TYPE(MPI_Group), INTENT(IN) :: group
29
         INTEGER, INTENT(OUT) :: rank
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_Group_size(group, size, ierror)
33
         TYPE(MPI_Group), INTENT(IN) :: group
34
         INTEGER, INTENT(OUT) :: size
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
     MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)
37
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
38
         INTEGER, INTENT(IN) :: n, ranks1(n)
39
         INTEGER, INTENT(OUT) :: ranks2(n)
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_Group_union(group1, group2, newgroup, ierror)
43
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
44
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
     MPI_Intercomm_create_from_groups(local_group, local_leader, remote_group,
47
                   remote_leader, stringtag, info, errhandler, newintercomm,
48
```

<pre>ierror) TYPE(MPI_Group), INTENT(IN) :: local_group, remote_group</pre>	1 2
INTEGER, INTENT(IN) :: local_leader, remote_leader	3
CHARACTER(LEN=*), INTENT(IN) :: stringtag	4
TYPE(MPI_Info), INTENT(IN) :: info	5
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	6
TYPE(MPI_Comm), INTENT(OUT) :: newintercomm	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8
MDI Intercomm amonto (local comm local locdor near comm nemeto locdor	9
<pre>MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm, ierror)</pre>	10
TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm	11
INTEGER, INTENT(IN) :: local_leader, remote_leader, tag	12
TYPE(MPI_Comm), INTENT(OUT) :: newintercomm	13
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	14
	15 16
MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)	17
TYPE(MPI_Comm), INTENT(IN) :: intercomm LOGICAL, INTENT(IN) :: high	18
TYPE(MPI_Comm), INTENT(OUT) :: newintracomm	19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
	21
<pre>MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,</pre>	22
extra_state, ierror)	23
PROCEDURE(MPI_Type_copy_attr_function), INTENT(IN) :: type_copy_attr_fn	24
PROCEDURE(MPI_Type_delete_attr_function), INTENT(IN) ::	25
<pre>type_delete_attr_fn INTEGER, INTENT(OUT) :: type_keyval</pre>	26
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state	27
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	28
	29
MPI_Type_delete_attr(datatype, type_keyval, ierror)	30 31
TYPE(MPI_Datatype), INTENT(IN) :: datatype	32
INTEGER, INTENT(IN) :: type_keyval	33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,	35
attribute_val_out, flag, ierror)	36
TYPE(MPI_Datatype) :: oldtype	37
INTEGER :: type_keyval, ierror	38
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,	39
attribute_val_out	40
LOGICAL :: flag	41
MPI_Type_free_keyval(type_keyval, ierror)	42
INTEGER, INTENT(INOUT) :: type_keyval	43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)	45 46
TYPE(MPI_Datatype), INTENT(IN) :: datatype	40 47
INTEGER, INTENT(IN) :: type_keyval	48
· · · · · · · · · · · · · · · · · · ·	10

```
1
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
2
         LOGICAL, INTENT(OUT) :: flag
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Type_get_name(datatype, type_name, resultlen, ierror)
5
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name
7
         INTEGER, INTENT(OUT) :: resultlen
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
     MPI_TYPE_NULL_COPY_FN(oldtype, type_keyval, extra_state, attribute_val_in,
11
                   attribute_val_out, flag, ierror)
12
         TYPE(MPI_Datatype) :: oldtype
13
         INTEGER :: type_keyval, ierror
14
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
15
                   attribute_val_out
16
         LOGICAL :: flag
17
     MPI_TYPE_NULL_DELETE_FN(datatype, type_keyval, attribute_val, extra_state,
18
                   ierror)
19
         TYPE(MPI_Datatype) :: datatype
20
         INTEGER :: type_keyval
21
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
22
         INTEGER, INTENT(OUT) :: ierror
23
24
     MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
25
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
         INTEGER, INTENT(IN) :: type_keyval
27
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_Type_set_name(datatype, type_name, ierror)
30
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
31
         CHARACTER(LEN=*), INTENT(IN) :: type_name
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
35
                   extra_state, ierror)
36
         PROCEDURE(MPI_Win_copy_attr_function), INTENT(IN) :: win_copy_attr_fn
37
         PROCEDURE(MPI_Win_delete_attr_function), INTENT(IN) ::
38
                   win_delete_attr_fn
39
         INTEGER, INTENT(OUT) :: win_keyval
40
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     MPI_Win_delete_attr(win, win_keyval, ierror)
43
         TYPE(MPI_Win), INTENT(IN) :: win
44
         INTEGER, INTENT(IN) :: win_keyval
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
48
```

MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,	1
attribute_val_out, flag, ierror)	2 3
TYPE(MPI_Win) :: oldwin	
INTEGER :: win_keyval, ierror	4
<pre>INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,</pre>	5
attribute_val_out	6
LOGICAL :: flag	7 8
MPI_Win_free_keyval(win_keyval, ierror)	9
INTEGER, INTENT(INOUT) :: win_keyval	9 10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
	11
<pre>MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)</pre>	12
TYPE(MPI_Win), INTENT(IN) :: win	14
INTEGER, INTENT(IN) :: win_keyval	15
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val	16
LOGICAL, INTENT(OUT) :: flag	17
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	18
MPI_Win_get_name(win, win_name, resultlen, ierror)	19
TYPE(MPI_Win), INTENT(IN) :: win	20
CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name	21
INTEGER, INTENT(OUT) :: resultlen	22
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	23
MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state, attribute_val_in,	24
attribute_val_out, flag, ierror)	25
TYPE(MPI_Win) :: oldwin	26
INTEGER :: win_keyval, ierror	27
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,	28
attribute_val_out	29
LOGICAL :: flag	30
	31
<pre>MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, extra_state, ierror)</pre>	32
TYPE(MPI_Win) :: win	33
INTEGER :: win_keyval, ierror	34
<pre>INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state</pre>	35
MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)	36
TYPE(MPI_Win), INTENT(IN) :: win	37
INTEGER, INTENT(IN) :: win_keyval	38
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val	39
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
MDT lin oct none(min min none ismos)	41
MPI_Win_set_name(win, win_name, ierror)	42
TYPE(MPI_Win), INTENT(IN) :: win	43
CHARACTER(LEN=*), INTENT(IN) :: win_name	44
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45
	46
	47
	48

```
1
     A.4.6 Process Topologies Fortran 2008 Bindings
\mathbf{2}
     MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         INTEGER, INTENT(IN) :: rank, maxdims
5
         INTEGER, INTENT(OUT) :: coords(maxdims)
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
9
         TYPE(MPI_Comm), INTENT(IN) :: comm_old
10
         INTEGER, INTENT(IN) :: ndims, dims(ndims)
11
         LOGICAL, INTENT(IN) :: periods(ndims), reorder
12
         TYPE(MPI_Comm), INTENT(OUT) :: comm_cart
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
    MPI_Cartdim_get(comm, ndims, ierror)
15
         TYPE(MPI_Comm), INTENT(IN) :: comm
16
         INTEGER, INTENT(OUT) :: ndims
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)
20
         TYPE(MPI_Comm), INTENT(IN) :: comm
21
         INTEGER, INTENT(IN) :: maxdims
22
         INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
23
         LOGICAL, INTENT(OUT) :: periods(maxdims)
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
    MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)
26
         TYPE(MPI_Comm), INTENT(IN) :: comm
27
         INTEGER, INTENT(IN) :: ndims, dims(ndims)
28
         LOGICAL, INTENT(IN) :: periods(ndims)
29
         INTEGER, INTENT(OUT) :: newrank
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
    MPI_Cart_rank(comm, coords, rank, ierror)
33
         TYPE(MPI_Comm), INTENT(IN) :: comm
34
         INTEGER, INTENT(IN) :: coords(*)
35
         INTEGER, INTENT(OUT) :: rank
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
    MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         INTEGER, INTENT(IN) :: direction, disp
40
         INTEGER, INTENT(OUT) :: rank_source, rank_dest
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
    MPI_Cart_sub(comm, remain_dims, newcomm, ierror)
44
         TYPE(MPI_Comm), INTENT(IN) :: comm
45
         LOGICAL, INTENT(IN) :: remain_dims(*)
46
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Dims_create(nnodes, ndims, dims, ierror)
                                                                                  2
    INTEGER, INTENT(IN) :: nnodes, ndims
    INTEGER, INTENT(INOUT) :: dims(ndims)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  5
MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
                                                                                  6
             outdegree, destinations, destweights, info, reorder,
             comm_dist_graph, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                  9
    INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*),
                                                                                  10
              outdegree, destinations(outdegree), destweights(*)
                                                                                  11
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  12
    LOGICAL, INTENT(IN) :: reorder
                                                                                  13
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
                                                                                  14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  15
MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
                                                                                  16
                                                                                  17
              info, reorder, comm_dist_graph, ierror)
                                                                                  18
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                  19
    INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*),
                                                                                  20
              weights(*)
                                                                                  21
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  22
    LOGICAL, INTENT(IN) :: reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
                                                                                  23
                                                                                  24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  25
MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
                                                                                  26
             maxoutdegree, destinations, destweights, ierror)
                                                                                  27
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  28
    INTEGER, INTENT(IN) :: maxindegree, maxoutdegree
                                                                                  29
    INTEGER, INTENT(OUT) :: sources(maxindegree),
                                                                                  30
              destinations(maxoutdegree)
                                                                                  31
    INTEGER :: sourceweights(*), destweights(*)
                                                                                  32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  33
                                                                                  34
MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
                                                                                  35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  36
    INTEGER, INTENT(OUT) :: indegree, outdegree
                                                                                  37
    LOGICAL, INTENT(OUT) :: weighted
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
                                                                                  40
              ierror)
                                                                                  41
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                  42
    INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
                                                                                  43
    LOGICAL, INTENT(IN) :: reorder
                                                                                  44
    TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
                                                                                  45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  46
                                                                                  47
MPI_Graphdims_get(comm, nnodes, nedges, ierror)
                                                                                  48
```

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         INTEGER, INTENT(OUT) :: nnodes, nedges
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
5
         TYPE(MPI_Comm), INTENT(IN) :: comm
6
         INTEGER, INTENT(IN) :: maxindex, maxedges
7
         INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
     MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror)
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
         INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
13
         INTEGER, INTENT(OUT) :: newrank
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
     MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror)
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         INTEGER, INTENT(IN) :: rank, maxneighbors
18
         INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror)
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         INTEGER, INTENT(IN) :: rank
24
         INTEGER, INTENT(OUT) :: nneighbors
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
27
                   recvtype, comm, request, ierror)
28
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
29
         INTEGER, INTENT(IN) :: sendcount, recvcount
30
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
31
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
32
         TYPE(MPI_Comm), INTENT(IN) :: comm
33
         TYPE(MPI_Request), INTENT(OUT) :: request
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
37
                   displs, recvtype, comm, request, ierror)
38
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
39
         INTEGER, INTENT(IN) :: sendcount
40
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
41
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
42
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         TYPE(MPI_Request), INTENT(OUT) :: request
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
     MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
47
                   recvtype, comm, request, ierror)
48
```

```
TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  1
                                                                                  2
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  5
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                  a
             recvcounts, rdispls, recvtype, comm, request, ierror)
                                                                                  10
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  11
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                  12
              recvcounts(*), rdispls(*)
                                                                                  13
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  14
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  16
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  18
                                                                                  19
MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
             recvcounts, rdispls, recvtypes, comm, request, ierror)
                                                                                  20
                                                                                  21
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  22
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                  23
                                                                                  24
              rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                  25
                                                                                  26
              recvtypes(*)
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  27
                                                                                  28
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  29
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  30
                                                                                  31
MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
                                                                                  32
             recvcount, recvtype, comm, info, request, ierror)
                                                                                  33
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  34
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  35
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  36
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  37
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  38
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  39
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  41
                                                                                  42
MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  43
             recvtype, comm, ierror)
                                                                                  44
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  45
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  46
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  47
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  48
```

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
4
                   recvcounts, displs, recvtype, comm, info, request, ierror)
5
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
6
         INTEGER, INTENT(IN) :: sendcount, displs(*)
7
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
8
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
9
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         TYPE(MPI_Info), INTENT(IN) :: info
12
         TYPE(MPI_Request), INTENT(OUT) :: request
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
    MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
16
                   displs, recvtype, comm, ierror)
17
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
18
         INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
19
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
20
         TYPE(*), DIMENSION(...) :: recvbuf
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,
24
                   recvcount, recvtype, comm, info, request, ierror)
25
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
26
         INTEGER, INTENT(IN) :: sendcount, recvcount
27
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
28
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         TYPE(MPI_Info), INTENT(IN) :: info
31
         TYPE(MPI_Request), INTENT(OUT) :: request
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
35
                  recvtype, comm, ierror)
36
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
37
         INTEGER, INTENT(IN) :: sendcount, recvcount
38
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
39
         TYPE(*), DIMENSION(...) :: recvbuf
40
         TYPE(MPI_Comm), INTENT(IN) :: comm
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
43
                   recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
44
                   ierror)
45
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
46
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
47
                   recvcounts(*), rdispls(*)
48
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   2
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   5
                                                                                   6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
             recvcounts, rdispls, recvtype, comm, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  10
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
                                                                                  11
              rdispls(*)
                                                                                  12
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  13
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                  14
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  16
                                                                                  17
MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
                                                                                  18
             recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
                                                                                  19
              ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  20
                                                                                  21
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
                                                                                  22
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                  23
              rdispls(*)
                                                                                  24
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                  25
              recvtypes(*)
                                                                                  26
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  27
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  28
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  29
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  30
                                                                                  31
MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                  32
             recvcounts, rdispls, recvtypes, comm, ierror)
                                                                                  33
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  34
    INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
                                                                                  35
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
                                                                                  36
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
                                                                                  37
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  38
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  39
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  40
                                                                                  41
MPI_Topo_test(comm, status, ierror)
                                                                                  42
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  43
    INTEGER, INTENT(OUT) :: status
                                                                                  44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  45
                                                                                  46
                                                                                  47
```

48

1 A.4.7 MPI Environmental Management Fortran 2008 Bindings $\mathbf{2}$ DOUBLE PRECISION MPI_Wtick() 3 4 DOUBLE PRECISION MPI_Wtime() 5MPI_Add_error_class(errorclass, ierror) 6 INTEGER, INTENT(OUT) :: errorclass 7 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8 9 MPI_Add_error_code(errorclass, errorcode, ierror) 10 INTEGER, INTENT(IN) :: errorclass 11 INTEGER, INTENT(OUT) :: errorcode 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 MPI_Add_error_string(errorcode, string, ierror) 14 INTEGER, INTENT(IN) :: errorcode 15CHARACTER(LEN=*), INTENT(IN) :: string 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17 18 MPI_Alloc_mem(size, info, baseptr, ierror) 19 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 20INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size 21TYPE(MPI_Info), INTENT(IN) :: info 22 TYPE(C_PTR), INTENT(OUT) :: baseptr 23INTEGER, OPTIONAL, INTENT(OUT) :: ierror 24MPI_Comm_call_errhandler(comm, errorcode, ierror) 25TYPE(MPI_Comm), INTENT(IN) :: comm 26INTEGER, INTENT(IN) :: errorcode 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2829MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror) 30 PROCEDURE(MPI_Comm_errhandler_function), INTENT(IN) :: 31comm_errhandler_fn 32 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 MPI_Comm_get_errhandler(comm, errhandler, ierror) 35 TYPE(MPI_Comm), INTENT(IN) :: comm 36 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 MPI_Comm_set_errhandler(comm, errhandler, ierror) 40TYPE(MPI_Comm), INTENT(IN) :: comm 41 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 MPI_Errhandler_free(errhandler, ierror) 44TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4647 MPI_Error_class(errorcode, errorclass, ierror) 48

INTEGER, INTENT(IN) :: errorcode	1
INTEGER, INTENT(OUT) :: errorclass	2
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	3
MPI_Error_string(errorcode, string, resultlen, ierror)	4
INTEGER, INTENT(IN) :: errorcode	5
CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string	6
INTEGER, INTENT(OUT) :: resultlen	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8
	9
MPI_File_call_errhandler(fh, errorcode, ierror)	10
TYPE(MPI_File), INTENT(IN) :: fh	11
INTEGER, INTENT(IN) :: errorcode	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13
MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror)	14 15
PROCEDURE(MPI_File_errhandler_function), INTENT(IN) ::	15
file_errhandler_fn	10
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	19
	20
MPI_File_get_errhandler(file, errhandler, ierror)	21
TYPE(MPI_File), INTENT(IN) :: file	22
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler INTEGER, OPTIONAL, INTENT(OUT) :: ierror	23
INIEGER, OPTIONAL, INIENI(001) .: Terror	24
MPI_File_set_errhandler(file, errhandler, ierror)	25
TYPE(MPI_File), INTENT(IN) :: file	26
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	27
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	28
MPI_Free_mem(base, ierror)	29
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: base	30
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	31
	32
<pre>MPI_Get_library_version(version, resultlen, ierror)</pre>	33
CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version	34
INTEGER, INTENT(OUT) :: resultlen	35
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	36
MPI_Get_processor_name(name, resultlen, ierror)	37
CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name	38
INTEGER, INTENT(OUT) :: resultlen	39
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
	41
MPI_Get_version(version, subversion, ierror)	42
INTEGER, INTENT(OUT) :: version, subversion	43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
MPI_Session_call_errhandler(session, errorcode, ierror)	45
TYPE(MPI_Session), INTENT(IN) :: session	46
INTEGER, INTENT(IN) :: errorcode	47
	48

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Session_create_errhandler(session_errhandler_fn, errhandler, ierror)
3
         PROCEDURE(MPI_Session_errhandler_function), INTENT(IN) ::
4
                    session_errhandler_fn
5
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Session_get_errhandler(session, errhandler, ierror)
9
         TYPE(MPI_Session), INTENT(IN) :: session
10
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_Session_set_errhandler(session, errhandler, ierror)
13
         TYPE(MPI_Session), INTENT(IN) :: session
14
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Win_call_errhandler(win, errorcode, ierror)
18
         TYPE(MPI_Win), INTENT(IN) :: win
19
         INTEGER, INTENT(IN) :: errorcode
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror)
22
         PROCEDURE(MPI_Win_errhandler_function), INTENT(IN) :: win_errhandler_fn
23
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     MPI_Win_get_errhandler(win, errhandler, ierror)
27
         TYPE(MPI_Win), INTENT(IN) :: win
28
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
     MPI_Win_set_errhandler(win, errhandler, ierror)
^{31}
         TYPE(MPI_Win), INTENT(IN) :: win
32
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
36
     A.4.8 The Info Object Fortran 2008 Bindings
37
38
     MPI_Info_create_env(info, ierror)
         TYPE(MPI_Info), INTENT(OUT) :: info
39
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
     MPI_Info_create(info, ierror)
42
         TYPE(MPI_Info), INTENT(OUT) :: info
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_Info_delete(info, key, ierror)
46
         TYPE(MPI_Info), INTENT(IN) :: info
47
         CHARACTER(LEN=*), INTENT(IN) :: key
48
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

1 MPI_Info_dup(info, newinfo, ierror) TYPE(MPI_Info), INTENT(IN) :: info 2 TYPE(MPI_Info), INTENT(OUT) :: newinfo INTEGER, OPTIONAL, INTENT(OUT) :: ierror 5 MPI_Info_free(info, ierror) 6 TYPE(MPI_Info), INTENT(INOUT) :: info INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Info_get(info, key, valuelen, value, flag, ierror) 10 TYPE(MPI_Info), INTENT(IN) :: info 11 CHARACTER(LEN=*), INTENT(IN) :: key INTEGER, INTENT(IN) :: valuelen 1213 CHARACTER(LEN=valuelen), INTENT(OUT) :: value 14LOGICAL, INTENT(OUT) :: flag 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 16MPI_Info_get_nkeys(info, nkeys, ierror) 17 TYPE(MPI_Info), INTENT(IN) :: info 18 INTEGER, INTENT(OUT) :: nkeys 19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2021MPI_Info_get_nthkey(info, n, key, ierror) 22 TYPE(MPI_Info), INTENT(IN) :: info 23INTEGER, INTENT(IN) :: n 24CHARACTER(LEN=*), INTENT(OUT) :: key 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 26MPI_Info_get_string(info, key, buflen, value, flag, ierror) 27TYPE(MPI_Info), INTENT(IN) :: info 28CHARACTER(LEN=*), INTENT(IN) :: key 29 INTEGER, INTENT(INOUT) :: buflen 30 CHARACTER(LEN=*), INTENT(OUT) :: value 31LOGICAL, INTENT(OUT) :: flag 32 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 33 34 MPI_Info_get_valuelen(info, key, valuelen, flag, ierror) 35TYPE(MPI_Info), INTENT(IN) :: info 36 CHARACTER(LEN=*), INTENT(IN) :: key 37 INTEGER, INTENT(OUT) :: valuelen 38 LOGICAL, INTENT(OUT) :: flag 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 MPI_Info_set(info, key, value, ierror) 41 TYPE(MPI_Info), INTENT(IN) :: info 42CHARACTER(LEN=*), INTENT(IN) :: key, value 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44 4546A.4.9 Process Creation and Management Fortran 2008 Bindings 47MPI_Abort(comm, errorcode, ierror) 48

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         INTEGER, INTENT(IN) :: errorcode
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Close_port(port_name, ierror)
5
         CHARACTER(LEN=*), INTENT(IN) :: port_name
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)
9
         CHARACTER(LEN=*), INTENT(IN) :: port_name
10
         TYPE(MPI_Info), INTENT(IN) :: info
11
         INTEGER, INTENT(IN) :: root
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
     MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)
16
         CHARACTER(LEN=*), INTENT(IN) :: port_name
17
         TYPE(MPI_Info), INTENT(IN) :: info
18
         INTEGER, INTENT(IN) :: root
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_Comm_disconnect(comm, ierror)
24
         TYPE(MPI_Comm), INTENT(INOUT) :: comm
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     MPI_Comm_get_parent(parent, ierror)
27
         TYPE(MPI_Comm), INTENT(OUT) :: parent
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_Comm_join(fd, intercomm, ierror)
31
         INTEGER, INTENT(IN) :: fd
32
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
35
                   array_of_errcodes, ierror)
36
         CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
37
         INTEGER, INTENT(IN) :: maxprocs, root
38
         TYPE(MPI_Info), INTENT(IN) :: info
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
41
         INTEGER :: array_of_errcodes(*)
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
45
                   array_of_maxprocs, array_of_info, root, comm, intercomm,
46
                   array_of_errcodes, ierror)
47
         INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
48
```

```
1
    CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
                                                                                   2
               array_of_argv(count, *)
    TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
    INTEGER :: array_of_errcodes(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Finalized(flag, ierror)
    LOGICAL, INTENT(OUT) :: flag
                                                                                   10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   11
MPI_Finalize(ierror)
                                                                                   12
                                                                                   13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   14
MPI_Initialized(flag, ierror)
                                                                                   15
    LOGICAL, INTENT(OUT) :: flag
                                                                                   16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   17
                                                                                   18
MPI_Init(ierror)
                                                                                   19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   20
MPI_Init_thread(required, provided, ierror)
                                                                                   21
    INTEGER, INTENT(IN) :: required
                                                                                   22
    INTEGER, INTENT(OUT) :: provided
                                                                                   23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   24
                                                                                   25
MPI_Is_thread_main(flag, ierror)
                                                                                   26
    LOGICAL, INTENT(OUT) :: flag
                                                                                   27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   28
MPI_Lookup_name(service_name, info, port_name, ierror)
                                                                                   29
    CHARACTER(LEN=*), INTENT(IN) :: service_name
                                                                                   30
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   31
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
                                                                                   32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   33
                                                                                   34
MPI_Open_port(info, port_name, ierror)
                                                                                   35
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   36
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
                                                                                   37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   38
MPI_Publish_name(service_name, info, port_name, ierror)
                                                                                   39
    CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
                                                                                   40
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   42
                                                                                   43
MPI_Query_thread(provided, ierror)
                                                                                   44
    INTEGER, INTENT(OUT) :: provided
                                                                                   45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   46
MPI_Session_finalize(session, ierror)
                                                                                   47
    TYPE(MPI_Session), INTENT(INOUT) :: session
                                                                                   48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Session_get_info(session, info_used, ierror)
3
         TYPE(MPI_Session), INTENT(IN) :: session
4
         TYPE(MPI_Info), INTENT(OUT) :: info_used
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7
     MPI_Session_get_nth_pset(session, info, n, pset_len, pset_name, ierror)
8
         TYPE(MPI_Session), INTENT(IN) :: session
9
         TYPE(MPI_Info), INTENT(IN) :: info
10
         INTEGER, INTENT(IN) :: n
11
         INTEGER, INTENT(INOUT) :: pset_len
12
         CHARACTER(LEN=*), INTENT(OUT) :: pset_name
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     MPI_Session_get_num_psets(session, info, npset_names, ierror)
15
         TYPE(MPI_Session), INTENT(IN) :: session
16
         TYPE(MPI_Info), INTENT(IN) :: info
17
         INTEGER, INTENT(OUT) :: npset_names
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     MPI_Session_get_pset_info(session, pset_name, info, ierror)
21
         TYPE(MPI_Session), INTENT(IN) :: session
22
         CHARACTER(LEN=*), INTENT(IN) :: pset_name
23
         TYPE(MPI_Info), INTENT(OUT) :: info
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     MPI_Session_init(info, errhandler, session, ierror)
26
         TYPE(MPI_Info), INTENT(IN) :: info
27
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
28
         TYPE(MPI_Session), INTENT(OUT) :: session
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
     MPI_Unpublish_name(service_name, info, port_name, ierror)
32
         CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
33
         TYPE(MPI_Info), INTENT(IN) :: info
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     A.4.10 One-Sided Communications Fortran 2008 Bindings
37
38
     MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
39
                   target_disp, target_count, target_datatype, op, win, ierror)
40
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
41
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
42
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
43
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
44
         TYPE(MPI_Op), INTENT(IN) :: op
45
         TYPE(MPI_Win), INTENT(IN) :: win
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

```
1
MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype,
                                                                                  2
              target_rank, target_disp, win, ierror)
                                                                                  3
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr,
              compare_addr
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
                                                                                  5
                                                                                  6
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: target_rank
                                                                                  7
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  8
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  9
                                                                                  10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  11
MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,
                                                                                  12
             target_disp, op, win, ierror)
                                                                                  13
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  14
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
                                                                                  15
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  16
    INTEGER, INTENT(IN) :: target_rank
                                                                                  17
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  18
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  19
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  20
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  21
MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
                                                                                  22
                                                                                  23
             result_count, result_datatype, target_rank, target_disp,
                                                                                  24
             target_count, target_datatype, op, win, ierror)
                                                                                  25
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  26
    INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
                                                                                  27
              target_count
                                                                                  28
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
                                                                                  29
              target_datatype
                                                                                  30
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
                                                                                  31
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  32
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  33
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  35
MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                  36
             target_disp, target_count, target_datatype, win, ierror)
                                                                                  37
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
                                                                                  38
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                  39
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                  40
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  41
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  43
                                                                                  44
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                  45
             target_disp, target_count, target_datatype, win, ierror)
                                                                                  46
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  47
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                  48
```

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
2
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
3
         TYPE(MPI_Win), INTENT(IN) :: win
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
6
                  target_disp, target_count, target_datatype, op, win, request,
7
                  ierror)
8
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
9
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
10
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
11
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
12
         TYPE(MPI_Op), INTENT(IN) :: op
13
         TYPE(MPI_Win), INTENT(IN) :: win
14
         TYPE(MPI_Request), INTENT(OUT) :: request
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
18
                  result_addr, result_count, result_datatype, target_rank,
19
                  target_disp, target_count, target_datatype, op, win, request,
20
                  ierror)
21
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
22
         INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
23
                   target_count
24
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
25
                   target_datatype
26
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
27
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
28
         TYPE(MPI_Op), INTENT(IN) :: op
29
         TYPE(MPI_Win), INTENT(IN) :: win
30
         TYPE(MPI_Request), INTENT(OUT) :: request
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
33
                  target_disp, target_count, target_datatype, win, request,
34
                  ierror)
35
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
36
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
37
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
38
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
39
         TYPE(MPI_Win), INTENT(IN) :: win
40
         TYPE(MPI_Request), INTENT(OUT) :: request
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
    MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,
44
                  target_disp, target_count, target_datatype, win, request,
45
                  ierror)
46
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
47
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
48
```

TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype 1 2 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Win), INTENT(IN) :: win TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size 9 INTEGER, INTENT(IN) :: disp_unit 10 TYPE(MPI_Info), INTENT(IN) :: info 11 TYPE(MPI_Comm), INTENT(IN) :: comm 12TYPE(C_PTR), INTENT(OUT) :: baseptr 13 TYPE(MPI_Win), INTENT(OUT) :: win 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1516MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror) 17 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 18 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size 19 INTEGER, INTENT(IN) :: disp_unit TYPE(MPI_Info), INTENT(IN) :: info 2021TYPE(MPI_Comm), INTENT(IN) :: comm 22 TYPE(C_PTR), INTENT(OUT) :: baseptr 23TYPE(MPI_Win), INTENT(OUT) :: win 24 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 25MPI_Win_attach(win, base, size, ierror) 26TYPE(MPI_Win), INTENT(IN) :: win 27TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base 28INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 31MPI_Win_complete(win, ierror) 32 TYPE(MPI_Win), INTENT(IN) :: win 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 MPI_Win_create(base, size, disp_unit, info, comm, win, ierror) 35 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base 36 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size 37 INTEGER, INTENT(IN) :: disp_unit 38 TYPE(MPI_Info), INTENT(IN) :: info 39 TYPE(MPI_Comm), INTENT(IN) :: comm 40 TYPE(MPI_Win), INTENT(OUT) :: win 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4243 MPI_Win_create_dynamic(info, comm, win, ierror) 44 TYPE(MPI_Info), INTENT(IN) :: info 45TYPE(MPI_Comm), INTENT(IN) :: comm 46TYPE(MPI_Win), INTENT(OUT) :: win 47INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

```
1
     MPI_Win_detach(win, base, ierror)
\mathbf{2}
         TYPE(MPI_Win), INTENT(IN) :: win
3
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Win_fence(assert, win, ierror)
6
         INTEGER, INTENT(IN) :: assert
7
         TYPE(MPI_Win), INTENT(IN) :: win
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
     MPI_Win_flush_all(win, ierror)
11
         TYPE(MPI_Win), INTENT(IN) :: win
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_Win_flush_local_all(win, ierror)
14
         TYPE(MPI_Win), INTENT(IN) :: win
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Win_flush_local(rank, win, ierror)
18
         INTEGER, INTENT(IN) :: rank
19
         TYPE(MPI_Win), INTENT(IN) :: win
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_Win_flush(rank, win, ierror)
22
         INTEGER, INTENT(IN) :: rank
23
         TYPE(MPI_Win), INTENT(IN) :: win
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     MPI_Win_free(win, ierror)
27
         TYPE(MPI_Win), INTENT(INOUT) :: win
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_Win_get_group(win, group, ierror)
30
         TYPE(MPI_Win), INTENT(IN) :: win
31
         TYPE(MPI_Group), INTENT(OUT) :: group
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Win_get_info(win, info_used, ierror)
35
         TYPE(MPI_Win), INTENT(IN) :: win
36
         TYPE(MPI_Info), INTENT(OUT) :: info_used
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_Win_lock_all(assert, win, ierror)
39
         INTEGER, INTENT(IN) :: assert
40
         TYPE(MPI_Win), INTENT(IN) :: win
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_Win_lock(lock_type, rank, assert, win, ierror)
44
         INTEGER, INTENT(IN) :: lock_type, rank, assert
45
         TYPE(MPI_Win), INTENT(IN) :: win
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
     MPI_Win_post(group, assert, win, ierror)
48
```

1 TYPE(MPI_Group), INTENT(IN) :: group 2 INTEGER, INTENT(IN) :: assert TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror 5 MPI_Win_set_info(win, info, ierror) 6 TYPE(MPI_Win), INTENT(IN) :: win TYPE(MPI_Info), INTENT(IN) :: info INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10 MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 11 TYPE(MPI_Win), INTENT(IN) :: win 1213 INTEGER, INTENT(IN) :: rank 14INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size 15INTEGER, INTENT(OUT) :: disp_unit 16TYPE(C_PTR), INTENT(OUT) :: baseptr 17 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 MPI_Win_start(group, assert, win, ierror) 19 TYPE(MPI_Group), INTENT(IN) :: group 20INTEGER, INTENT(IN) :: assert 21TYPE(MPI_Win), INTENT(IN) :: win 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2324MPI_Win_sync(win, ierror) 25TYPE(MPI_Win), INTENT(IN) :: win 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 27MPI_Win_test(win, flag, ierror) 28 TYPE(MPI_Win), INTENT(IN) :: win 29 LOGICAL, INTENT(OUT) :: flag 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 31 32 MPI_Win_unlock_all(win, ierror) 33 TYPE(MPI_Win), INTENT(IN) :: win 34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35MPI_Win_unlock(rank, win, ierror) 36 INTEGER, INTENT(IN) :: rank 37 TYPE(MPI_Win), INTENT(IN) :: win 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 40 MPI_Win_wait(win, ierror) 41 TYPE(MPI_Win), INTENT(IN) :: win 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 44 A.4.11 External Interfaces Fortran 2008 Bindings 4546MPI_Grequest_complete(request, ierror) 47TYPE(MPI_Request), INTENT(IN) :: request 48

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
3
                   ierror)
4
         PROCEDURE(MPI_Grequest_query_function), INTENT(IN) :: query_fn
5
         PROCEDURE(MPI_Grequest_free_function), INTENT(IN) :: free_fn
6
         PROCEDURE(MPI_Grequest_cancel_function), INTENT(IN) :: cancel_fn
7
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
8
         TYPE(MPI_Request), INTENT(OUT) :: request
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_Status_set_cancelled(status, flag, ierror)
12
         TYPE(MPI_Status), INTENT(INOUT) :: status
13
         LOGICAL, INTENT(IN) :: flag
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
     MPI_Status_set_elements(status, datatype, count, ierror)
16
         TYPE(MPI_Status), INTENT(INOUT) :: status
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         INTEGER, INTENT(IN) :: count
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Status_set_elements_x(status, datatype, count, ierror)
22
         TYPE(MPI_Status), INTENT(INOUT) :: status
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
     A.4.12 I/O Fortran 2008 Bindings
28
29
     MPI_CONVERSION_FN_NULL(userbuf, datatype, count, filebuf, position,
30
                   extra_state, ierror)
31
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
32
         TYPE(C_PTR), VALUE :: userbuf, filebuf
33
         TYPE(MPI_Datatype) :: datatype
34
         INTEGER :: count, ierror
35
         INTEGER(KIND=MPI_OFFSET_KIND) :: position
36
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
37
     MPI_File_close(fh, ierror)
38
         TYPE(MPI_File), INTENT(INOUT) :: fh
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_File_delete(filename, info, ierror)
42
         CHARACTER(LEN=*), INTENT(IN) :: filename
43
         TYPE(MPI_Info), INTENT(IN) :: info
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_File_get_amode(fh, amode, ierror)
46
47
         TYPE(MPI_File), INTENT(IN) :: fh
48
         INTEGER, INTENT(OUT) :: amode
```

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2 MPI_File_get_atomicity(fh, flag, ierror) TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag 5 INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_byte_offset(fh, offset, disp, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 10 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp 11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 12MPI_File_get_group(fh, group, ierror) 13 TYPE(MPI_File), INTENT(IN) :: fh 14 TYPE(MPI_Group), INTENT(OUT) :: group 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617 MPI_File_get_info(fh, info_used, ierror) 18 TYPE(MPI_File), INTENT(IN) :: fh 19 TYPE(MPI_Info), INTENT(OUT) :: info_used 20INTEGER, OPTIONAL, INTENT(OUT) :: ierror 21MPI_File_get_position(fh, offset, ierror) 22 TYPE(MPI_File), INTENT(IN) :: fh 23INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2526MPI_File_get_position_shared(fh, offset, ierror) 27TYPE(MPI_File), INTENT(IN) :: fh 28 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 MPI_File_get_size(fh, size, ierror) 31TYPE(MPI_File), INTENT(IN) :: fh 32 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 35MPI_File_get_type_extent(fh, datatype, extent, ierror) 36 TYPE(MPI_File), INTENT(IN) :: fh 37 TYPE(MPI_Datatype), INTENT(IN) :: datatype 38 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror) 41 TYPE(MPI_File), INTENT(IN) :: fh 42INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp 43 TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype 44 CHARACTER(LEN=*), INTENT(OUT) :: datarep 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4647MPI_File_iread_all(fh, buf, count, datatype, request, ierror) 48

```
1
         TYPE(MPI_File), INTENT(IN) :: fh
2
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
3
         INTEGER, INTENT(IN) :: count
4
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
5
         TYPE(MPI_Request), INTENT(OUT) :: request
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
    MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror)
8
         TYPE(MPI_File), INTENT(IN) :: fh
9
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
10
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
11
         INTEGER, INTENT(IN) :: count
12
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
13
         TYPE(MPI_Request), INTENT(OUT) :: request
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
17
         TYPE(MPI_File), INTENT(IN) :: fh
18
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
19
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
20
         INTEGER, INTENT(IN) :: count
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         TYPE(MPI_Request), INTENT(OUT) :: request
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_File_iread(fh, buf, count, datatype, request, ierror)
25
         TYPE(MPI_File), INTENT(IN) :: fh
26
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
27
         INTEGER, INTENT(IN) :: count
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Request), INTENT(OUT) :: request
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)
33
         TYPE(MPI_File), INTENT(IN) :: fh
34
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
35
         INTEGER, INTENT(IN) :: count
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         TYPE(MPI_Request), INTENT(OUT) :: request
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
40
         TYPE(MPI_File), INTENT(IN) :: fh
41
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
42
         INTEGER, INTENT(IN) :: count
43
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
44
         TYPE(MPI_Request), INTENT(OUT) :: request
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_File_iwrite_at_all(fh, offset, buf, count, datatype, request, ierror)
48
```

TYPE(MPI_File), INTENT(IN) :: fh 1 2 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count 4 TYPE(MPI_Datatype), INTENT(IN) :: datatype 5 TYPE(MPI_Request), INTENT(OUT) :: request 6 INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror) TYPE(MPI_File), INTENT(IN) :: fh 10 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 11 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 12INTEGER, INTENT(IN) :: count 13 TYPE(MPI_Datatype), INTENT(IN) :: datatype 14TYPE(MPI_Request), INTENT(OUT) :: request 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617 MPI_File_iwrite(fh, buf, count, datatype, request, ierror) 18 TYPE(MPI_File), INTENT(IN) :: fh 19 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count 2021TYPE(MPI_Datatype), INTENT(IN) :: datatype 22 TYPE(MPI_Request), INTENT(OUT) :: request 23INTEGER, OPTIONAL, INTENT(OUT) :: ierror 24MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror) 25TYPE(MPI_File), INTENT(IN) :: fh 26TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 27INTEGER, INTENT(IN) :: count 28TYPE(MPI_Datatype), INTENT(IN) :: datatype 29 TYPE(MPI_Request), INTENT(OUT) :: request 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3132 MPI_File_open(comm, filename, amode, info, fh, ierror) 33 TYPE(MPI_Comm), INTENT(IN) :: comm 34 CHARACTER(LEN=*), INTENT(IN) :: filename 35INTEGER, INTENT(IN) :: amode 36 TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_File), INTENT(OUT) :: fh 37 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 MPI_File_preallocate(fh, size, ierror) 40 TYPE(MPI_File), INTENT(IN) :: fh 41 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 44MPI_File_read_all_begin(fh, buf, count, datatype, ierror) 45TYPE(MPI_File), INTENT(IN) :: fh 46TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 47INTEGER, INTENT(IN) :: count 48

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_File_read_all_end(fh, buf, status, ierror)
4
         TYPE(MPI_File), INTENT(IN) :: fh
5
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
6
         TYPE(MPI_Status) :: status
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
     MPI_File_read_all(fh, buf, count, datatype, status, ierror)
10
         TYPE(MPI_File), INTENT(IN) :: fh
11
         TYPE(*), DIMENSION(..) :: buf
12
         INTEGER, INTENT(IN) :: count
13
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
         TYPE(MPI_Status) :: status
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
     MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
17
         TYPE(MPI_File), INTENT(IN) :: fh
18
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
19
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
20
         INTEGER, INTENT(IN) :: count
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
     MPI_File_read_at_all_end(fh, buf, status, ierror)
25
         TYPE(MPI_File), INTENT(IN) :: fh
26
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
27
         TYPE(MPI_Status) :: status
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
30
         TYPE(MPI_File), INTENT(IN) :: fh
31
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
32
         TYPE(*), DIMENSION(..) :: buf
33
         INTEGER, INTENT(IN) :: count
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
         TYPE(MPI_Status) :: status
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
39
         TYPE(MPI_File), INTENT(IN) :: fh
40
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
41
         TYPE(*), DIMENSION(..) :: buf
42
         INTEGER, INTENT(IN) :: count
43
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
44
         TYPE(MPI_Status) :: status
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
     MPI_File_read(fh, buf, count, datatype, status, ierror)
47
         TYPE(MPI_File), INTENT(IN) :: fh
48
```

```
1
    TYPE(*), DIMENSION(..) :: buf
                                                                                   2
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   5
MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
                                                                                   10
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   12
                                                                                   13
MPI_File_read_ordered_end(fh, buf, status, ierror)
                                                                                   14
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   15
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                   16
    TYPE(MPI_Status) :: status
                                                                                   17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   18
MPI_File_read_ordered(fh, buf, count, datatype, status, ierror)
                                                                                   19
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  20
    TYPE(*), DIMENSION(..) :: buf
                                                                                  21
    INTEGER, INTENT(IN) :: count
                                                                                  22
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  23
    TYPE(MPI_Status) :: status
                                                                                  24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  25
                                                                                  26
MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
                                                                                  27
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  28
    TYPE(*), DIMENSION(..) :: buf
                                                                                  29
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  30
                                                                                   31
    TYPE(MPI_Status) :: status
                                                                                   32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  33
MPI_File_seek(fh, offset, whence, ierror)
                                                                                  34
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  35
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  36
    INTEGER, INTENT(IN) :: whence
                                                                                  37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  38
                                                                                   39
MPI_File_seek_shared(fh, offset, whence, ierror)
                                                                                   40
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   41
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  42
    INTEGER, INTENT(IN) :: whence
                                                                                  43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   44
MPI_File_set_atomicity(fh, flag, ierror)
                                                                                   45
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   46
    LOGICAL, INTENT(IN) :: flag
                                                                                   47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   48
```

```
1
    MPI_File_set_info(fh, info, ierror)
\mathbf{2}
         TYPE(MPI_File), INTENT(IN) :: fh
3
         TYPE(MPI_Info), INTENT(IN) :: info
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_File_set_size(fh, size, ierror)
6
         TYPE(MPI_File), INTENT(IN) :: fh
7
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
     MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)
11
         TYPE(MPI_File), INTENT(IN) :: fh
12
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
13
         TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype
14
         CHARACTER(LEN=*), INTENT(IN) :: datarep
15
         TYPE(MPI_Info), INTENT(IN) :: info
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_File_sync(fh, ierror)
18
         TYPE(MPI_File), INTENT(IN) :: fh
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
22
         TYPE(MPI_File), INTENT(IN) :: fh
23
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
24
         INTEGER, INTENT(IN) :: count
25
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_File_write_all_end(fh, buf, status, ierror)
28
         TYPE(MPI_File), INTENT(IN) :: fh
29
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
30
         TYPE(MPI_Status) :: status
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_File_write_all(fh, buf, count, datatype, status, ierror)
34
         TYPE(MPI_File), INTENT(IN) :: fh
35
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
36
         INTEGER, INTENT(IN) :: count
37
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
         TYPE(MPI_Status) :: status
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
41
         TYPE(MPI_File), INTENT(IN) :: fh
42
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
43
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
44
         INTEGER, INTENT(IN) :: count
45
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

```
MPI_File_write_at_all_end(fh, buf, status, ierror)
                                                                                   1
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   2
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   5
MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                   10
    INTEGER, INTENT(IN) :: count
                                                                                   11
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   12
    TYPE(MPI_Status) :: status
                                                                                   13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  14
                                                                                   15
MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
                                                                                   16
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   17
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                   18
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                   19
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   20
                                                                                  21
    TYPE(MPI_Status) :: status
                                                                                  22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  23
MPI_File_write(fh, buf, count, datatype, status, ierror)
                                                                                   ^{24}
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  25
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                   26
    INTEGER, INTENT(IN) :: count
                                                                                  27
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  28
    TYPE(MPI_Status) :: status
                                                                                  29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  30
                                                                                  31
MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
                                                                                  32
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  33
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  34
    INTEGER, INTENT(IN) :: count
                                                                                  35
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  37
MPI_File_write_ordered_end(fh, buf, status, ierror)
                                                                                  38
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  39
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                   40
    TYPE(MPI_Status) :: status
                                                                                  41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  42
                                                                                  43
MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
                                                                                  44
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   45
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                   46
    INTEGER, INTENT(IN) :: count
                                                                                   47
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   48
```

```
1
         TYPE(MPI_Status) :: status
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_File_write_shared(fh, buf, count, datatype, status, ierror)
4
         TYPE(MPI_File), INTENT(IN) :: fh
5
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
6
         INTEGER, INTENT(IN) :: count
7
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
         TYPE(MPI_Status) :: status
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
12
                   dtype_file_extent_fn, extra_state, ierror)
13
         CHARACTER(LEN=*), INTENT(IN) :: datarep
14
         PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn,
15
                   write_conversion_fn
16
         PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
17
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     A.4.13 Language Bindings Fortran 2008 Bindings
21
22
     MPI_F_sync_reg(buf)
23
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
24
     MPI_Status_f082f(f08_status, f_status, ierror)
25
         TYPE(MPI_Status), INTENT(IN) :: f08_status
26
         INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     MPI_Status_f2f08(f_status, f08_status, ierror)
30
         INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
31
         TYPE(MPI_Status), INTENT(OUT) :: f08_status
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Type_create_f90_complex(p, r, newtype, ierror)
         INTEGER, INTENT(IN) :: p, r
35
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Type_create_f90_integer(r, newtype, ierror)
39
         INTEGER, INTENT(IN) :: r
40
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     MPI_Type_create_f90_real(p, r, newtype, ierror)
43
44
         INTEGER, INTENT(IN) :: p, r
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
45
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
     MPI_Type_match_size(typeclass, size, datatype, ierror)
48
```

```
1
    INTEGER, INTENT(IN) :: typeclass, size
                                                                                        \mathbf{2}
    TYPE(MPI_Datatype), INTENT(OUT) :: datatype
                                                                                        3
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        4
                                                                                        5
A.4.14 Tools / Profiling Interface Fortran 2008 Bindings
                                                                                        6
                                                                                        7
MPI_Pcontrol(level)
    INTEGER, INTENT(IN) :: level
                                                                                        9
                                                                                        10
A.4.15 Deprecated Fortran 2008 Bindings
                                                                                        11
                                                                                        12
MPI_Info_get(info, key, valuelen, value, flag, ierror)
                                                                                        13
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                        14
    CHARACTER(LEN=*), INTENT(IN) :: key
                                                                                        15
    INTEGER, INTENT(IN) :: valuelen
                                                                                        16
    CHARACTER(LEN=valuelen), INTENT(OUT) :: value
                                                                                        17
    LOGICAL, INTENT(OUT) :: flag
                                                                                        18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        19
MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
                                                                                        20
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                       21
    CHARACTER(LEN=*), INTENT(IN) :: key
                                                                                       22
    INTEGER, INTENT(OUT) :: valuelen
                                                                                       23
    LOGICAL, INTENT(OUT) :: flag
                                                                                        ^{24}
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        25
                                                                                        26
MPI_Sizeof(x, size, ierror)
                                                                                       27
    TYPE(*), DIMENSION(..) :: x
                                                                                       28
    INTEGER, INTENT(OUT) :: size
                                                                                       29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       30
                                                                                        31
                                                                                        32
                                                                                        33
                                                                                       34
                                                                                       35
                                                                                       36
                                                                                       37
                                                                                        38
                                                                                        39
                                                                                        40
                                                                                        41
                                                                                       42
                                                                                        43
                                                                                        44
                                                                                        45
                                                                                        46
                                                                                        47
                                                                                        48
```

A.5 Fortran Bindings with mpif.h or the mpi Module 1 $\mathbf{2}$ A.5.1 Point-to-Point Communication Fortran Bindings 3 4 MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 5<type> BUF(*) 6 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 7 MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 8 <type> BUF(*) 9 10 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 11 MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR) 12<type> BUFFER(*) 13 INTEGER SIZE, IERROR 1415MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR) 16<type> BUFFER_ADDR(*) 17INTEGER SIZE, IERROR 18 MPI_CANCEL(REQUEST, IERROR) 19 INTEGER REQUEST, IERROR 2021MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR) 22INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR 23MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 24<type> BUF(*) 25INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 2627MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR) 28INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR 29LOGICAL FLAG 30 MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR) 31 <type> BUF(*) 32 INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR 33 34MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR) 35 INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR 36 LOGICAL FLAG 37 MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) 38 <type> BUF(*) 39 INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR 4041 MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 42<type> BUF(*) 43 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 44MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 45<type> BUF(*) 46 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 47 48

MPI_ISENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,	1
COMM, REQUEST, IERROR)	2
<type> BUF(*)</type>	3
INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, REQUEST,	4
IERROR	5
	6
MPI_ISENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,	7
RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, REQUEST, IERROR)	8
<type> SENDBUF(*), RECVBUF(*)</type>	9
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,	10
SOURCE, RECVTAG, COMM, REQUEST, IERROR	11
MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	12
<pre><type> BUF(*)</type></pre>	13
	14
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	15
MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR)	16
INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR	17
	18
MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)	19
<type> BUF(*)</type>	
INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR	20
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)	21
INTEGER SOURCE, TAG, COMM, STATUS, TELLION, INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR	22
INTEGER SUBROE, TRG, COMM, STRIDS(MP1_STRIDS_SIZE), TERROR	23
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)	24
<type> BUF(*)</type>	25
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),	26
IERROR	27
	28
MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)	29
<type> BUF(*)</type>	30
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR	31
MPI_REQUEST_FREE(REQUEST, IERROR)	32
INTEGER REQUEST, IERROR	33
	34
MPI_REQUEST_GET_STATUS(REQUEST, FLAG, STATUS, IERROR)	35
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR	36
LOGICAL FLAG	37
	38
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	39
<type> BUF(*)</type>	40
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	41
MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	42
<pre><type> BUF(*)</type></pre>	43
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	44
	45
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	46
<type> BUF(*)</type>	47
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	48

```
1
    MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
\mathbf{2}
         <type> BUF(*)
3
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
4
     MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
5
                   COMM, STATUS, IERROR)
6
         <type> BUF(*)
7
         INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
8
                   STATUS(MPI_STATUS_SIZE), IERROR
9
10
     MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
11
                   RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
12
         <type> SENDBUF(*), RECVBUF(*)
13
         INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
14
                   SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
15
     MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
16
         <type> BUF(*)
17
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
18
19
     MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
20
         <type> BUF(*)
21
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
22
     MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
23
         INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
24
25
    MPI_START(REQUEST, IERROR)
26
         INTEGER REQUEST, IERROR
27
     MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
28
         INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,
29
                   *), IERROR
30
         LOGICAL FLAG
31
32
     MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
33
         INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
34
                   IERROR
35
         LOGICAL FLAG
36
     MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
37
         INTEGER STATUS (MPI_STATUS_SIZE), IERROR
38
         LOGICAL FLAG
39
40
     MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
41
         INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
42
         LOGICAL FLAG
43
    MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
44
                   ARRAY_OF_STATUSES, IERROR)
45
         INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
46
                   ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR
47
48
```

FFI_WATTALE(COUNT, ARRAT_OF_REQUESTS, ARRAT_OF_STATUSES, TERROR)	1
INTEGEN COUNT, AMMAT_DF_REQUESTS(*), AMMAT_DF_STATUSES(MT_STATUS_STZE,	2
*), IEROR	3 4
MPT WATTANY(COUNT, ARRAY OF REQUESTS, INDEX, STATUS, IERROR)	5
	6
IERROR	7
MPI_WAIT(REQUEST, STATUS, IERROR)	8
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR	9
MOT MATTSOME (INCOUNT ADDAY OF DECHESTS OUTCOINT ADDAY OF INDICES	10
APPAV OF STATUSES (FERDOR)	11 12
INTECED INCOUNT ADDAY OF DECHESTS(+) OUTCOUNT ADDAY OF INDICES(+)	12
ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR	14
I	15
A.5.2 Partitioned Communication Fortran Bindings	16
1	17
	18 19
	19 20
	21
MPI_PREADY_LIST(LENGTH, ARRAY_OF_PARTITIONS, REQUEST, IERROR)	22
INTEGER LENGTH, ARRAY_OF_PARTITIONS(*), REQUEST, IERROR	23
	24
	25
MPI_PREADY_RANGE(PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR)	26 27
INTEGER PARTITION LOW, PARTITION HIGH, REQUEST, IERROR	28
MPI_PRECV_INIT(BUF, PARTITIONS, COUNT, DATATYPE, DEST, TAG, COMM, INFO,	29
REQUEST, IERROR)	30
(types bor())	31
	32
	33 34
MPI_PSEND_INIT(BUF, PARTITIONS, COUNT, DATATYPE, DEST, TAG, COMM, INFO,	35
REQUEST, IERROR)	36
<type> BUF(*)</type>	37
INTEGER PARTITIONS, DATATYPE, DEST, TAG, COMM, INFO, REQUEST, IERROR INTEGER(KIND=MPI_COUNT_KIND) COUNT	38
INTEGER(KIND-PIPI_COUNI_KIND) COUNT	39
	40
A.5.3 Datatypes Fortran Bindings	$41 \\ 42$
INTECED (VIND-MDI ADDRESS VIND) MDI AINT ADD (DASE DISD)	42
INTECED (VIND-MDI ADDRESS VIND) DASE DISD	44
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_DIFF(ADDR1, ADDR2)	45
	46
4	47
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)	48

1 2 3	<type> LOCATION(*) INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS INTEGER IERROR</type>
4 5 6	MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
7 8 9 10	MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) COUNT
10 11 12 13 14 15 16	<pre>MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,</pre>
17 18 19 20 21	MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR) CHARACTER*(*) DATAREP INTEGER INCOUNT, DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
21 22 23 24	<pre>MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)</pre>
25 26 27	MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR) INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
28 29	MPI_TYPE_COMMIT(DATATYPE, IERROR) INTEGER DATATYPE, IERROR
30 31 32	MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
33 34 35 36 37 38	<pre>MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES, ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER, OLDTYPE, NEWTYPE, IERROR) INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*), ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR</pre>
39 40 41 42 43	MPI_TYPE_CREATE_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
44 45 46 47 48	<pre>MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,</pre>

MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,	1
IERROR)	2
INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR	3
INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE	4
MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,	5
OLDTYPE, NEWTYPE, IERROR)	6
INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,	7 8
NEWTYPE, IERROR	° 9
	9 10
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)	10
INTEGER OLDTYPE, NEWTYPE, IERROR	12
INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT	13
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,	14
ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)	15
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,	16
IERROR	17
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)	18
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,	19
ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)	20
INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),	21
ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR	22
	23
MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR)	24
INTEGER OLDTYPE, NEWTYPE, IERROR	25
MPI_TYPE_FREE(DATATYPE, IERROR)	26
INTEGER DATATYPE, IERROR	27
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	28
ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,	29 30
IERROR)	30 31
INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	32
ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR	33
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)	34
	35
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,	36
COMBINER, IERROR)	37
INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,	38
IERROR	39
MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)	40
INTEGER DATATYPE, IERROR	41
INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT	42
MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR)	43
INTEGER DATATYPE, IERROR	44
INTEGER (KIND=MPI_COUNT_KIND) LB, EXTENT	45
	46
MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)	47
INTEGER DATATYPE, IERROR	48

```
1
         INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
\mathbf{2}
     MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
3
         INTEGER DATATYPE, IERROR
4
         INTEGER(KIND=MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT
5
6
     MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
7
                   OLDTYPE, NEWTYPE, IERROR)
8
         INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
9
                   OLDTYPE, NEWTYPE, IERROR
10
     MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
11
         INTEGER DATATYPE, SIZE, IERROR
12
13
    MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)
14
         INTEGER DATATYPE, IERROR
15
         INTEGER(KIND=MPI_COUNT_KIND) SIZE
16
    MPI TYPE VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
17
         INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
18
19
     MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
20
                   DATATYPE, IERROR)
21
         CHARACTER*(*) DATAREP
22
         <type> INBUF(*), OUTBUF(*)
23
         INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
24
         INTEGER OUTCOUNT, DATATYPE, IERROR
25
     MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
26
                   IERROR)
27
         <type> INBUF(*), OUTBUF(*)
28
         INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
29
30
31
     A.5.4 Collective Communication Fortran Bindings
32
     MPI_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
33
34
                   RECVTYPE, COMM, INFO, REQUEST, IERROR)
         <type> SENDBUF(*), RECVBUF(*)
35
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
36
37
                   IERROR
38
     MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
39
                   COMM, IERROR)
40
         <type> SENDBUF(*), RECVBUF(*)
41
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
42
     MPI_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
43
44
                   DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
45
         <type> SENDBUF(*), RECVBUF(*)
46
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
47
                   INFO, REQUEST, IERROR
48
```

MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, IERROR)	$\frac{1}{2}$
<pre><type> SENDBUF(*), RECVBUF(*)</type></pre>	3
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	4
IERROR	5
MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO,	6 7
REQUEST, IERROR)	8
<type> SENDBUF(*), RECVBUF(*)</type>	9
INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR	10
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)	11
<type> SENDBUF(*), RECVBUF(*)</type>	12
INTEGER COUNT, DATATYPE, OP, COMM, IERROR	13
MPI_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	14
RECVTYPE, COMM, INFO, REQUEST, IERROR)	15 16
<type> SENDBUF(*), RECVBUF(*)</type>	10
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,	18
IERROR	19
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	20
COMM, IERROR)	21
<type> SENDBUF(*), RECVBUF(*)</type>	22
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR	23
	24
MPI_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)	25
<pre><type> SENDBUF(*), RECVBUF(*)</type></pre>	26
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),	27 28
RECVTYPE, COMM, INFO, REQUEST, IERROR	20 29
	30
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,	31
RDISPLS, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*)</type>	32
<pre>SENDBOR(*), RECVEOR(*) INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),</pre>	33
RECVTYPE, COMM, IERROR	34
	35
MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,	36
RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR)	37
<type> SENDBUF(*), RECVBUF(*)</type>	38
<pre>INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*), RDISPLS(*), RECVTYPES(*), COMM, INFO, REQUEST, IERROR</pre>	39
RDISPLS(*), RECVITPES(*), COMM, INFO, REQUEST, TERROR	40
MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,	41 42
RDISPLS, RECVTYPES, COMM, IERROR)	43
<type> SENDBUF(*), RECVBUF(*)</type>	44
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),	45
RDISPLS(*), RECVTYPES(*), COMM, IERROR	46
MPI_BARRIER(COMM, IERROR)	47
INTEGER COMM, IERROR	48

1 MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR) $\mathbf{2}$ INTEGER COMM, INFO, REQUEST, IERROR 3 MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR) 4 <type> BUFFER(*) 5INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR 6 $\overline{7}$ MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR) 8 <type> BUFFER(*) 9 INTEGER COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR 10 MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, 11 IERROR) 12<type> SENDBUF(*), RECVBUF(*) 13 INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 1415MPI_EXSCAN (SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) 16<type> SENDBUF(*), RECVBUF(*) 17 INTEGER COUNT, DATATYPE, OP, COMM, IERROR 18 MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 19 ROOT, COMM, INFO, REQUEST, IERROR) 20<type> SENDBUF(*), RECVBUF(*) 21INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, 22 REQUEST, IERROR 2324MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 25ROOT, COMM, IERROR) 26<type> SENDBUF(*), RECVBUF(*) 27INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR 28MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 29 RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR) 30 <type> SENDBUF(*), RECVBUF(*) 31INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, 32 COMM, INFO, REQUEST, IERROR 33 34MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 35 RECVTYPE, ROOT, COMM, IERROR) 36 <type> SENDBUF(*), RECVBUF(*) 37 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, 38 COMM, IERROR 39 MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 40COMM, REQUEST, IERROR) 41 <type> SENDBUF(*), RECVBUF(*) 42INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 43 44MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 45 RECVTYPE, COMM, REQUEST, IERROR) 46 <type> SENDBUF(*), RECVBUF(*) 4748

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, REQUEST, IERROR	1 2
MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)	3 4 5
<type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR</type>	6 7
<pre>MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR</type></pre>	8 9 10 11
<pre>MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,</pre>	12 13 14 15 16 17
<pre>MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,</pre>	18 19 20 21 22 23
MPI_IBARRIER(COMM, REQUEST, IERROR) INTEGER COMM, REQUEST, IERROR	23 24 25
<pre>MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)</pre>	26 27 28 29
<pre>MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)</pre>	30 31 32
<pre>MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR</type></pre>	33 34 35 36 37 38
<pre>MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,</pre>	 39 40 41 42 43
<pre>MPI_IREDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,</pre>	44 45 46 47 48

1MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, $\mathbf{2}$ REQUEST, IERROR) 3 <type> SENDBUF(*), RECVBUF(*) 4 INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR 5MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, 6 IERROR) 7 <type> SENDBUF(*), RECVBUF(*) 8 INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR 9 10MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) 11 <type> SENDBUF(*), RECVBUF(*) 12INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 13 MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 14 ROOT, COMM, REQUEST, IERROR) 15<type> SENDBUF(*), RECVBUF(*) 16 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, 17 IERROR 18 19 MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, 20RECVTYPE, ROOT, COMM, REQUEST, IERROR) 21<type> SENDBUF(*), RECVBUF(*) 22INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, 23COMM, REQUEST, IERROR 24MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR) 25INTEGER OP, IERROR 26LOGICAL COMMUTE 2728MPI_OP_CREATE(USER_FN, COMMUTE, OP, IERROR) 29EXTERNAL USER_FN 30 LOGICAL COMMUTE 31INTEGER OP, IERROR 32 MPI_OP_FREE(OP, IERROR) 33 INTEGER OP, IERROR 34 35 MPI_REDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO, 36 REQUEST, IERROR) 37 <type> SENDBUF(*), RECVBUF(*) 38 INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR 39 MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR) 40<type> INBUF(*), INOUTBUF(*) 41 INTEGER COUNT, DATATYPE, OP, IERROR 4243 MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, 44 COMM, INFO, REQUEST, IERROR) 45<type> SENDBUF(*), RECVBUF(*) 46INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 47 48

MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, IERROR)	1 2
<type> SENDBUF(*), RECVBUF(*)</type>	3
INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR	5
MPI_REDUCE_SCATTER_INIT(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, INFO, REQUEST, IERROR)	6
<pre><type> SENDBUF(*), RECVBUF(*)</type></pre>	7
INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, INFO, REQUEST, IERROR	8 9
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, IERROR)	10 11
<type> SENDBUF(*), RECVBUF(*)</type>	12
INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR	13
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)	14 15
<type> SENDBUF(*), RECVBUF(*)</type>	16
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR	17
MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,	18
IERROR)	19
<type> SENDBUF(*), RECVBUF(*)</type>	20 21
INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR	22
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)	23
<type> SENDBUF(*), RECVBUF(*)</type>	24
INTEGER COUNT, DATATYPE, OP, COMM, IERROR	25
MPI_SCATTER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	26 27
RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)	21
<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,</type>	29
REQUEST, IERROR	30
	31
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)	32
<pre><type> SENDBUF(*), RECVBUF(*)</type></pre>	33 34
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR	35
MPI_SCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,	36
RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)	37
<type> SENDBUF(*), RECVBUF(*)</type>	38
INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,	39 40
COMM, INFO, REQUEST, IERROR	40 41
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,	42
RECVTYPE, ROOT, COMM, IERROR)	43
<type> SENDBUF(*), RECVBUF(*)</type>	44
INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,	45
COMM, IERROR	46 47
	47

```
1
     A.5.5 Groups, Contexts, Communicators, and Caching Fortran Bindings
\mathbf{2}
     MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
3
         INTEGER COMM1, COMM2, RESULT, IERROR
4
5
     MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
6
         INTEGER COMM, GROUP, NEWCOMM, IERROR
7
     MPI_COMM_CREATE_FROM_GROUP(GROUP, STRINGTAG, INFO, ERRHANDLER, NEWCOMM,
8
                   IERROR)
9
         INTEGER GROUP, INFO, ERRHANDLER, NEWCOMM, IERROR
10
         CHARACTER*(*) STRINGTAG
11
12
     MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR)
13
         INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR
14
     MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
15
                   EXTRA_STATE, IERROR)
16
         EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
17
         INTEGER COMM_KEYVAL, IERROR
18
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
19
20
     MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
21
         INTEGER COMM, COMM_KEYVAL, IERROR
22
     MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
23
         INTEGER COMM, NEWCOMM, IERROR
^{24}
25
     MPI_COMM_DUP_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
26
                   ATTRIBUTE_VAL_OUT, FLAG, IERROR)
27
         INTEGER OLDCOMM, COMM_KEYVAL, IERROR
28
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
29
                    ATTRIBUTE_VAL_OUT
30
         LOGICAL FLAG
31
     MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)
32
         INTEGER COMM, INFO, NEWCOMM, IERROR
33
34
     MPI_COMM_FREE(COMM, IERROR)
35
         INTEGER COMM, IERROR
36
     MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
37
         INTEGER COMM_KEYVAL, IERROR
38
39
     MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
40
         INTEGER COMM, COMM_KEYVAL, IERROR
41
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
42
         LOGICAL FLAG
43
     MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR)
44
         INTEGER COMM, INFO_USED, IERROR
45
46
     MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
47
         INTEGER COMM, RESULTLEN, IERROR
48
```

1 CHARACTER*(*) COMM_NAME 2 MPI_COMM_GROUP(COMM, GROUP, IERROR) INTEGER COMM, GROUP, IERROR MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR) INTEGER COMM, NEWCOMM, REQUEST, IERROR MPI_COMM_IDUP_WITH_INFO(COMM, INFO, NEWCOMM, REQUEST, IERROR) INTEGER COMM, INFO, NEWCOMM, REQUEST, IERROR 10 MPI_COMM_NULL_COPY_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, 11 ATTRIBUTE_VAL_OUT, FLAG, IERROR) 12INTEGER OLDCOMM, COMM_KEYVAL, IERROR 13 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, 14ATTRIBUTE_VAL_OUT 15LOGICAL FLAG 16MPI_COMM_NULL_DELETE_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, 17IERROR) 18 INTEGER COMM, COMM_KEYVAL, IERROR 19 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE 2021MPI_COMM_RANK(COMM, RANK, IERROR) 22 INTEGER COMM, RANK, IERROR 23MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR) 24 INTEGER COMM, GROUP, IERROR 2526MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR) 27INTEGER COMM, SIZE, IERROR 28MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR) 29 INTEGER COMM, COMM_KEYVAL, IERROR 30 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL 3132 MPI_COMM_SET_INFO(COMM, INFO, IERROR) 33 INTEGER COMM, INFO, IERROR 34 MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR) 35INTEGER COMM, IERROR 36 CHARACTER*(*) COMM_NAME 37 38 MPI_COMM_SIZE(COMM, SIZE, IERROR) 39 INTEGER COMM, SIZE, IERROR 40 MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR) 41 INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR 4243 MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR) 44 INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR 45MPI_COMM_TEST_INTER(COMM, FLAG, IERROR) 46INTEGER COMM, IERROR 4748 LOGICAL FLAG

1 MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR) $\mathbf{2}$ INTEGER GROUP1, GROUP2, RESULT, IERROR 3 MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR) 4 INTEGER GROUP1, GROUP2, NEWGROUP, IERROR 56 MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR) 7 INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR 8 MPI_GROUP_FREE(GROUP, IERROR) 9 INTEGER GROUP, IERROR 10 11 MPI_GROUP_FROM_SESSION_PSET(SESSION, PSET_NAME, NEWGROUP, IERROR) 12INTEGER SESSION, NEWGROUP, IERROR 13CHARACTER*(*) PSET_NAME 14MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR) 15INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR 1617MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR) 18 INTEGER GROUP1, GROUP2, NEWGROUP, IERROR 19 MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR) 20INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR 2122MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR) 23INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR 24 MPI_GROUP_RANK(GROUP, RANK, IERROR) 25INTEGER GROUP, RANK, IERROR 2627MPI_GROUP_SIZE(GROUP, SIZE, IERROR) 28INTEGER GROUP, SIZE, IERROR 29 MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR) 30 INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR 3132 MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR) 33 INTEGER GROUP1, GROUP2, NEWGROUP, IERROR 34 MPI_INTERCOMM_CREATE_FROM_GROUPS(LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP, 35REMOTE_LEADER, STRINGTAG, INFO, ERRHANDLER, NEWINTERCOMM, 36 37 IERROR) INTEGER LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP, REMOTE_LEADER, INFO, 38 ERRHANDLER, NEWINTERCOMM, IERROR 39 40 CHARACTER*(*) STRINGTAG 41 MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, 42TAG, NEWINTERCOMM, IERROR) 43 INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG, 44 NEWINTERCOMM, IERROR 4546MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR) 47INTEGER INTERCOMM, NEWINTRACOMM, IERROR 48 LOGICAL HIGH

MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,	1
EXTRA_STATE, IERROR)	2
EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN	3
INTEGER TYPE_KEYVAL, IERROR	4
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	5
MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)	6
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	7
INTEGER DATATIFE, TIFE_REIVAL, TERROR	8
MPI_TYPE_DUP_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	9
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	10
INTEGER OLDTYPE, TYPE_KEYVAL, IERROR	11
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	12
ATTRIBUTE_VAL_OUT	13
LOGICAL FLAG	14
	15
MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)	16
INTEGER TYPE_KEYVAL, IERROR	17
MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	18
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	19
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	20
LOGICAL FLAG	21
	22
MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR)	23
INTEGER DATATYPE, RESULTLEN, IERROR	24
CHARACTER*(*) TYPE_NAME	25
MPI_TYPE_NULL_COPY_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	26
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	27
INTEGER OLDTYPE, TYPE_KEYVAL, IERROR	28
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	29
ATTRIBUTE_VAL_OUT	30
LOGICAL FLAG	31
	32
MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,	33
IERROR)	34
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	35
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	36
MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)	37
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	38
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	39
	40
MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)	41
INTEGER DATATYPE, IERROR	42
CHARACTER*(*) TYPE_NAME	43
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,	44
EXTRA_STATE, IERROR)	45
EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN	46
INTEGER WIN_KEYVAL, IERROR	47
THE GER WINTER THE AND TRUCK TO THE STATE OF	48

```
1
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
\mathbf{2}
     MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
3
         INTEGER WIN, WIN_KEYVAL, IERROR
4
5
     MPI_WIN_DUP_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
6
                   ATTRIBUTE_VAL_OUT, FLAG, IERROR)
7
         INTEGER OLDWIN, WIN_KEYVAL, IERROR
8
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
9
                    ATTRIBUTE_VAL_OUT
10
         LOGICAL FLAG
11
     MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
12
         INTEGER WIN_KEYVAL, IERROR
13
14
     MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
15
         INTEGER WIN, WIN_KEYVAL, IERROR
16
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
17
         LOGICAL FLAG
18
     MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
19
         INTEGER WIN, RESULTLEN, IERROR
20
         CHARACTER*(*) WIN_NAME
21
22
     MPI_WIN_NULL_COPY_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
23
                   ATTRIBUTE_VAL_OUT, FLAG, IERROR)
^{24}
         INTEGER OLDWIN, WIN_KEYVAL, IERROR
25
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
26
                    ATTRIBUTE_VAL_OUT
27
         LOGICAL FLAG
28
     MPI_WIN_NULL_DELETE_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
29
         INTEGER WIN, WIN_KEYVAL, IERROR
30
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
^{31}
32
     MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
33
         INTEGER WIN, WIN_KEYVAL, IERROR
34
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
35
     MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
36
         INTEGER WIN, IERROR
37
         CHARACTER*(*) WIN_NAME
38
39
40
     A.5.6 Process Topologies Fortran Bindings
41
42
     MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
         INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
43
44
     MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
45
         INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
46
         LOGICAL PERIODS(*), REORDER
47
48
     MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
```

INTEGER COMM, NDIMS, IERROR MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR) INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR LOGICAL PERIODS(*) MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR) INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR LOGICAL PERIODS(*) MPI_CART_RANK(COMM, COORDS, RANK, IERROR) 10 INTEGER COMM, COORDS(*), RANK, IERROR 11 MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR) 1213 INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR 14MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR) 15INTEGER COMM, NEWCOMM, IERROR 16LOGICAL REMAIN_DIMS(*) 1718 MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR) 19 INTEGER NNODES, NDIMS, DIMS(*), IERROR 20MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS, 21OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER, 22 COMM_DIST_GRAPH, IERROR) 23INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE, 24DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, 25IERROR 26LOGICAL REORDER 2728 MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS, 29 INFO, REORDER, COMM_DIST_GRAPH, IERROR) 30 INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*), 31WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR 32 LOGICAL REORDER 33 MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS, 34 MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR) 35INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE, 36 DESTINATIONS(*), DESTWEIGHTS(*), IERROR 37 38 MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR) 39 INTEGER COMM, INDEGREE, OUTDEGREE, IERROR 40LOGICAL WEIGHTED 41 MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH, 42IERROR) 43 INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR 44LOGICAL REORDER 4546MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR) 47

INTEGER COMM, NNODES, NEDGES, IERROR

1

2

48

1 MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR) $\mathbf{2}$ INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR 3 MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) 4 INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR 56 MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR) 7 INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR 8 MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR) 9 INTEGER COMM, RANK, NNEIGHBORS, IERROR 10 11 MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 12RECVTYPE, COMM, REQUEST, IERROR) 13<type> SENDBUF(*), RECVBUF(*) 14INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 15MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 16 DISPLS, RECVTYPE, COMM, REQUEST, IERROR) 17 <type> SENDBUF(*), RECVBUF(*) 18 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 19 REQUEST, IERROR 2021MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 22 RECVTYPE, COMM, REQUEST, IERROR) 23<type> SENDBUF(*), RECVBUF(*) 24INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 25MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, 26RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) 27<type> SENDBUF(*), RECVBUF(*) 28INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 29 RECVTYPE, COMM, REQUEST, IERROR 30 31MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 32 RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR) 33 <type> SENDBUF(*), RECVBUF(*) 34 INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM, 35REQUEST, IERROR 36 INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*) 37 MPI_NEIGHBOR_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, 38 RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR) 39 <type> SENDBUF(*), RECVBUF(*) 40INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 41 IERROR 4243 MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 44 RECVTYPE, COMM, IERROR) 45<type> SENDBUF(*), RECVBUF(*) 46INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 47 48

MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,	1
RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)	2
<type> SENDBUF(*), RECVBUF(*)</type>	3
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	4
	5
INFO, REQUEST, IERROR	
MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,	6
DISPLS, RECVTYPE, COMM, IERROR)	7
	8
<type> SENDBUF(*), RECVBUF(*)</type>	9
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	10
IERROR	11
MPI_NEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,	12
	13
RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)	14
<type> SENDBUF(*), RECVBUF(*)</type>	
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,	15
IERROR	16
	17
MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	18
RECVTYPE, COMM, IERROR)	19
<type> SENDBUF(*), RECVBUF(*)</type>	20
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR	21
	22
MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,	23
RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST,	24
IERROR)	25
<type> SENDBUF(*), RECVBUF(*)</type>	
<pre>INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),</pre>	26
RECVTYPE, COMM, INFO, REQUEST, IERROR	27
	28
MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,	29
RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)	30
<type> SENDBUF(*), RECVBUF(*)</type>	31
<pre>INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),</pre>	32
RECVTYPE, COMM, IERROR	33
	34
MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES,	35
RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST,	36
IERROR)	
<type> SENDBUF(*), RECVBUF(*)</type>	37
<pre>INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,</pre>	38
INFO, REQUEST, IERROR	39
<pre>INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)</pre>	40
	41
MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,	42
RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)	43
<type> SENDBUF(*), RECVBUF(*)</type>	44
<pre>INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,</pre>	45
IERROR	46
<pre>INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)</pre>	47
	48
	40

```
1
     MPI_TOPO_TEST(COMM, STATUS, IERROR)
\mathbf{2}
         INTEGER COMM, STATUS, IERROR
3
4
     A.5.7 MPI Environmental Management Fortran Bindings
5
6
     DOUBLE PRECISION MPI_WTICK()
7
     DOUBLE PRECISION MPI_WTIME()
8
9
     MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
10
         INTEGER ERRORCLASS, IERROR
11
     MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
12
         INTEGER ERRORCLASS, ERRORCODE, IERROR
13
14
     MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
15
         INTEGER ERRORCODE, IERROR
16
         CHARACTER*(*) STRING
17
     MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
18
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
19
         INTEGER INFO, IERROR
20
21
     If the Fortran compiler provides TYPE(C_PTR), then overloaded by:
22
       INTERFACE MPI_ALLOC_MEM
23
         SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
24
           IMPORT :: MPI_ADDRESS_KIND
25
           INTEGER :: INFO, IERROR
26
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
27
         END SUBROUTINE
28
         SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
29
           USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
30
           IMPORT :: MPI_ADDRESS_KIND
31
           INTEGER :: INFO, IERROR
32
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
33
           TYPE(C_PTR) :: BASEPTR
34
         END SUBROUTINE
35
       END INTERFACE
36
37
     MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
         INTEGER COMM, ERRORCODE, IERROR
38
39
     MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
40
         EXTERNAL COMM_ERRHANDLER_FN
41
         INTEGER ERRHANDLER, IERROR
42
     MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
43
44
         INTEGER COMM, ERRHANDLER, IERROR
45
     MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
46
         INTEGER COMM, ERRHANDLER, IERROR
47
48
     MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
```

INTEGER ERRHANDLER, IERROR	1
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)	2
INTEGER ERRORCODE, ERRORCLASS, IERROR	$\frac{3}{4}$
	4 5
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR) INTEGER ERRORCODE, RESULTLEN, IERROR	6
CHARACTER*(*) STRING	7
	8
MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)	9
INTEGER FH, ERRORCODE, IERROR	10
MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)	11
EXTERNAL FILE_ERRHANDLER_FN	12
INTEGER ERRHANDLER, IERROR	13 14
MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)	14
INTEGER FILE, ERRHANDLER, IERROR	16
MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)	17
INTEGER FILE, ERRHANDLER, IERROR	18
	19
MPI_FREE_MEM(BASE, IERROR)	20
<type> BASE(*)</type>	21
INTEGER IERROR	22
MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR)	23
CHARACTER*(*) VERSION	24 25
INTEGER RESULTLEN, IERROR	26
MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)	27
CHARACTER*(*) NAME	28
INTEGER RESULTLEN, IERROR	29
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)	30
INTEGER VERSION, SUBVERSION, IERROR	31
	32
MPI_SESSION_CALL_ERRHANDLER(SESSION, ERRORCODE, IERROR)	33
INTEGER SESSION, ERRORCODE, IERROR	34 35
MPI_SESSION_CREATE_ERRHANDLER(SESSION_ERRHANDLER_FN, ERRHANDLER, IERROR)	36
EXTERNAL SESSION_ERRHANDLER_FN	37
INTEGER ERRHANDLER, IERROR	38
MPI_SESSION_GET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)	39
INTEGER SESSION, ERRHANDLER, IERROR	40
MPI_SESSION_SET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)	41
INTEGER SESSION, ERRHANDLER, IERROR	42
	43
MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)	44 45
INTEGER WIN, ERRORCODE, IERROR	40
MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)	47
EXTERNAL WIN_ERRHANDLER_FN	48

```
1
         INTEGER ERRHANDLER, IERROR
\mathbf{2}
     MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
3
         INTEGER WIN, ERRHANDLER, IERROR
4
\mathbf{5}
     MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
6
         INTEGER WIN, ERRHANDLER, IERROR
7
8
     A.5.8 The Info Object Fortran Bindings
9
10
     MPI_INFO_CREATE_ENV(INFO, IERROR)
11
         INTEGER INFO, IERROR
12
     MPI_INFO_CREATE(INFO, IERROR)
13
         INTEGER INFO, IERROR
14
15
     MPI_INFO_DELETE(INFO, KEY, IERROR)
16
         INTEGER INFO, IERROR
17
         CHARACTER*(*) KEY
18
     MPI_INFO_DUP(INFO, NEWINFO, IERROR)
19
         INTEGER INFO, NEWINFO, IERROR
20
21
     MPI_INFO_FREE(INFO, IERROR)
22
         INTEGER INFO, IERROR
23
     MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
^{24}
25
         INTEGER INFO, VALUELEN, IERROR
26
         CHARACTER*(*) KEY, VALUE
27
         LOGICAL FLAG
28
     MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
29
         INTEGER INFO, NKEYS, IERROR
30
^{31}
     MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
32
         INTEGER INFO, N, IERROR
33
         CHARACTER*(*) KEY
34
     MPI_INFO_GET_STRING(INFO, KEY, BUFLEN, VALUE, FLAG, IERROR)
35
         INTEGER INFO, BUFLEN, IERROR
36
         CHARACTER*(*) KEY, VALUE
37
         LOGICAL FLAG
38
39
     MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
40
         INTEGER INFO, VALUELEN, IERROR
41
         CHARACTER*(*) KEY
42
         LOGICAL FLAG
43
     MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
44
         INTEGER INFO, IERROR
45
         CHARACTER*(*) KEY, VALUE
46
47
48
```

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	927
A.5.9 Process Creation and Management Fortran Bindings	1
MPI_ABORT(COMM, ERRORCODE, IERROR)	2
INTEGER COMM, ERRORCODE, IERROR	3
	4
MPI_CLOSE_PORT(PORT_NAME, IERROR) CHARACTER*(*) PORT_NAME	6
INTEGER IERROR	7
NDT COMM ACCEDT (DODT NAME INFO DOOT COMM NEUCOMM IEDDOD)	8
<pre>MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)</pre>	9
INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR	10 11
MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)	11
CHARACTER*(*) PORT_NAME	13
INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR	14
MPI_COMM_DISCONNECT(COMM, IERROR)	15
INTEGER COMM, IERROR	16 17
	18
MPI_COMM_GET_PARENT(PARENT, IERROR) INTEGER PARENT, IERROR	19
	20
MPI_COMM_JOIN(FD, INTERCOMM, IERROR) INTEGER FD, INTERCOMM, IERROR	21
	22 23
MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,	23
ARRAY_OF_ERRCODES, IERROR) CHARACTER*(*) COMMAND, ARGV(*)	25
INTEGER MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),	26
IERROR	27
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,	28 29
ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,	30
ARRAY_OF_ERRCODES, IERROR)	31
INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*), ROOT, COMM,	32
INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)	33
CHARACIER*(*) ARRAI_UF_CUMMANDS(*), ARRAI_UF_ARGV(CUUNI, *)	34 35
MPI_FINALIZED(FLAG, IERROR)	36
LOGICAL FLAG INTEGER IERROR	37
INTEGER TERROR	38
MPI_FINALIZE(IERROR)	39
INTEGER IERROR	40 41
MPI_INITIALIZED(FLAG, IERROR)	41 42
LOGICAL FLAG	43
INTEGER IERROR	44
MPI_INIT(IERROR)	45
INTEGER IERROR	46 47
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)	47 48

```
1
         INTEGER REQUIRED, PROVIDED, IERROR
\mathbf{2}
     MPI_IS_THREAD_MAIN(FLAG, IERROR)
3
         LOGICAL FLAG
4
         INTEGER IERROR
5
6
     MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
7
         CHARACTER*(*) SERVICE_NAME, PORT_NAME
8
         INTEGER INFO, IERROR
9
     MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
10
         INTEGER INFO, IERROR
11
         CHARACTER*(*) PORT_NAME
12
13
     MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
14
         CHARACTER*(*) SERVICE_NAME, PORT_NAME
15
         INTEGER INFO, IERROR
16
     MPI_QUERY_THREAD(PROVIDED, IERROR)
17
         INTEGER PROVIDED, IERROR
18
19
     MPI_SESSION_FINALIZE(SESSION, IERROR)
20
         INTEGER SESSION, IERROR
21
     MPI_SESSION_GET_INFO(SESSION, INFO_USED, IERROR)
22
         INTEGER SESSION, INFO_USED, IERROR
23
24
     MPI_SESSION_GET_NTH_PSET(SESSION, INFO, N, PSET_LEN, PSET_NAME, IERROR)
25
         INTEGER SESSION, INFO, N, PSET_LEN, IERROR
26
         CHARACTER*(*) PSET_NAME
27
     MPI_SESSION_GET_NUM_PSETS(SESSION, INFO, NPSET_NAMES, IERROR)
28
         INTEGER SESSION, INFO, NPSET_NAMES, IERROR
29
30
     MPI_SESSION_GET_PSET_INFO(SESSION, PSET_NAME, INFO, IERROR)
31
         INTEGER SESSION, INFO, IERROR
32
         CHARACTER*(*) PSET_NAME
33
34
     MPI_SESSION_INIT(INFO, ERRHANDLER, SESSION, IERROR)
         INTEGER INFO, ERRHANDLER, SESSION, IERROR
35
36
     MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
37
         CHARACTER*(*) SERVICE_NAME, PORT_NAME
38
         INTEGER INFO, IERROR
39
40
41
     A.5.10 One-Sided Communications Fortran Bindings
42
     MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
43
                   TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
44
         <type> ORIGIN_ADDR(*)
45
         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
46
                    TARGET_DATATYPE, OP, WIN, IERROR
47
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
48
```

MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,	1
TARGET_RANK, TARGET_DISP, WIN, IERROR)	2
<type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)</type>	3
INTEGER DATATYPE, TARGET_RANK, WIN, IERROR	4
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	5
	6
MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,	7
TARGET_DISP, OP, WIN, IERROR)	8
<type> ORIGIN_ADDR(*), RESULT_ADDR(*)</type>	9
INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR	10
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	11
MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,	12
RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,	13
TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)	14
<type> ORIGIN_ADDR(*), RESULT_ADDR(*)</type>	15
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,	16
TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR	17
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	18
	19
MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	20
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)	21
<type> ORIGIN_ADDR(*)</type>	22
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	23
TARGET_DATATYPE, WIN, IERROR	24
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	25
MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	26
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)	27
<pre><type> ORIGIN_ADDR(*)</type></pre>	28
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	29
TARGET_DATATYPE, WIN, IERROR	30
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	31
INTEGER (KIND-MFT_ADDRESS_KIND) TARGET_DISP	32
MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	33
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,	34
IERROR)	35
<type> ORIGIN_ADDR(*)</type>	36
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	37
TARGET_DATATYPE, OP, WIN, REQUEST, IERROR	38
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	39
	40
MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,	41
RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,	42
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,	43
IERROR)	44
<pre><type> ORIGIN_ADDR(*), RESULT_ADDR(*) INTEGED ORIGIN_COUNT_ORIGIN_DATATVDE_RESULT_COUNT_RESULT_DATATVDE</type></pre>	45
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,	46
TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,	47
IERROR	48

1	INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
2	MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
3	TARGET_DISP, TARGET_COUNT, TARGET_DATATIVE, WIN, REQUEST,
4	IERROR)
5	<pre><type> ORIGIN_ADDR(*)</type></pre>
6	INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
7	TARGET_DATATYPE, WIN, REQUEST, IERROR
8	INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
9	
10	MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
11	TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
12	IERROR)
13	<type> ORIGIN_ADDR(*)</type>
14	INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
15 16	TARGET_DATATYPE, WIN, REQUEST, IERROR
10	INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
18	MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
19	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
20	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
21	If the Fortran compiler provides TYPE(C_PTR), then overloaded by:
22	INTERFACE MPI_WIN_ALLOCATE_SHARED
23	SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
24	BASEPTR, WIN, IERROR)
25	IMPORT :: MPI_ADDRESS_KIND
26	INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
27	INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
28	END SUBROUTINE
29	SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
30	BASEPTR, WIN, IERROR)
31	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
32	IMPORT :: MPI_ADDRESS_KIND
33	INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
34	INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
35 36	TYPE(C_PTR) :: BASEPTR
37	END SUBROUTINE
38	END INTERFACE
39	MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
40	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
41	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
42	If the Fortran compiler provides TYPE(C_PTR), then overloaded by:
43	INTERFACE MPI_WIN_ALLOCATE
44	SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
45	WIN, IERROR)
46	IMPORT :: MPI_ADDRESS_KIND
47	INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
48	INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR

```
1
    END SUBROUTINE
    SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
                                                                                    2
          WIN, IERROR)
      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                    4
      IMPORT :: MPI_ADDRESS_KIND
                                                                                     5
                                                                                     6
      INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
      INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
                                                                                     7
      TYPE(C_PTR) :: BASEPTR
                                                                                     9
    END SUBROUTINE
                                                                                    10
  END INTERFACE
                                                                                    11
MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)
                                                                                    12
    INTEGER WIN, IERROR
                                                                                    13
    <type> BASE(*)
                                                                                    14
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
                                                                                    15
                                                                                    16
MPI_WIN_COMPLETE(WIN, IERROR)
                                                                                    17
    INTEGER WIN, IERROR
                                                                                    18
MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
                                                                                    19
    <type> BASE(*)
                                                                                    20
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
                                                                                    21
    INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
                                                                                    22
                                                                                    23
MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)
                                                                                    24
    INTEGER INFO, COMM, WIN, IERROR
                                                                                    25
MPI_WIN_DETACH(WIN, BASE, IERROR)
                                                                                    26
    INTEGER WIN, IERROR
                                                                                    27
    <type> BASE(*)
                                                                                    28
                                                                                    29
MPI_WIN_FENCE(ASSERT, WIN, IERROR)
                                                                                    30
    INTEGER ASSERT, WIN, IERROR
                                                                                    31
MPI_WIN_FLUSH_ALL(WIN, IERROR)
                                                                                    32
    INTEGER WIN, IERROR
                                                                                    33
                                                                                    34
MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
                                                                                    35
    INTEGER WIN, IERROR
                                                                                    36
MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
                                                                                    37
    INTEGER RANK, WIN, IERROR
                                                                                    38
                                                                                    39
MPI_WIN_FLUSH(RANK, WIN, IERROR)
                                                                                    40
    INTEGER RANK, WIN, IERROR
                                                                                    41
                                                                                    42
MPI_WIN_FREE(WIN, IERROR)
    INTEGER WIN, IERROR
                                                                                    43
                                                                                    44
MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
                                                                                    45
    INTEGER WIN, GROUP, IERROR
                                                                                    46
                                                                                    47
MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)
                                                                                    48
    INTEGER WIN, INFO_USED, IERROR
```

```
1
     MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
\mathbf{2}
         INTEGER ASSERT, WIN, IERROR
3
     MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
4
         INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR
5
6
     MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)
7
         INTEGER GROUP, ASSERT, WIN, IERROR
8
     MPI_WIN_SET_INFO(WIN, INFO, IERROR)
9
         INTEGER WIN, INFO, IERROR
10
11
     MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)
12
         INTEGER WIN, RANK, DISP_UNIT, IERROR
13
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
14
     If the Fortran compiler provides TYPE(C_PTR), then overloaded by:
15
       INTERFACE MPI_WIN_SHARED_QUERY
16
         SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &
17
               BASEPTR, IERROR)
18
           IMPORT :: MPI_ADDRESS_KIND
19
           INTEGER :: WIN, RANK, DISP_UNIT, IERROR
20
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
21
         END SUBROUTINE
22
         SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &
23
               BASEPTR, IERROR)
24
           USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
25
           IMPORT :: MPI_ADDRESS_KIND
26
           INTEGER :: WIN, RANK, DISP_UNIT, IERROR
27
           INTEGER(KIND=MPI_ADDRESS_KIND) ::
                                                SIZE
28
           TYPE(C_PTR) :: BASEPTR
29
         END SUBROUTINE
30
       END INTERFACE
^{31}
32
     MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
33
         INTEGER GROUP, ASSERT, WIN, IERROR
34
     MPI_WIN_SYNC(WIN, IERROR)
35
         INTEGER WIN, IERROR
36
37
     MPI_WIN_TEST(WIN, FLAG, IERROR)
38
         INTEGER WIN, IERROR
39
         LOGICAL FLAG
40
     MPI_WIN_UNLOCK_ALL(WIN, IERROR)
41
         INTEGER WIN, IERROR
42
43
     MPI_WIN_UNLOCK(RANK, WIN, IERROR)
44
         INTEGER RANK, WIN, IERROR
45
     MPI_WIN_WAIT(WIN, IERROR)
46
         INTEGER WIN, IERROR
47
48
```

A.5.11 External Interfaces Fortran Bindings 1 2 MPI_GREQUEST_COMPLETE(REQUEST, IERROR) INTEGER REQUEST, IERROR MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST, IERROR) EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE INTEGER REQUEST, IERROR 10 MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR) 11 INTEGER STATUS(MPI_STATUS_SIZE), IERROR 12LOGICAL FLAG 13 14MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR) 15INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR 16MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR) 17 INTEGER STATUS (MPI_STATUS_SIZE), DATATYPE, IERROR 18 INTEGER(KIND=MPI_COUNT_KIND) COUNT 19 2021A.5.12 I/O Fortran Bindings 22 MPI_CONVERSION_FN_NULL(USERBUF, DATATYPE, COUNT, FILEBUF, POSITION, 23EXTRA_STATE, IERROR) 24 <TYPE> USERBUF(*), FILEBUF(*) 25INTEGER DATATYPE, COUNT, IERROR 26INTEGER(KIND=MPI_OFFSET_KIND) POSITION 27INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE 28 29 MPI_FILE_CLOSE(FH, IERROR) 30 INTEGER FH, IERROR 31 MPI_FILE_DELETE(FILENAME, INFO, IERROR) 32 CHARACTER*(*) FILENAME 33 INTEGER INFO, IERROR 34 35MPI_FILE_GET_AMODE(FH, AMODE, IERROR) 36 INTEGER FH, AMODE, IERROR 37 MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) 38 INTEGER FH, IERROR 39 LOGICAL FLAG 40 41 MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR) 42INTEGER FH, IERROR 43 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP 44 45MPI_FILE_GET_GROUP(FH, GROUP, IERROR) 46INTEGER FH, GROUP, IERROR 47MPI_FILE_GET_INFO(FH, INFO_USED, IERROR) 48

1	INTEGER FH, INFO_USED, IERROR
2 3	MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)
4	INTEGER FH, IERROR
5	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
6	MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)
7	INTEGER FH, IERROR
8 9	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
10	MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
11	INTEGER FH, IERROR
12	INTEGER(KIND=MPI_OFFSET_KIND) SIZE
13	MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)
14 15	INTEGER FH, DATATYPE, IERROR
16	INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT
17	MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
18	INTEGER FH, ETYPE, FILETYPE, IERROR
19	INTEGER(KIND=MPI_OFFSET_KIND) DISP CHARACTER*(*) DATAREP
20	
21 22	MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
23	INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR <type> BUF(*)</type>
24	
25	MPI_FILE_IREAD_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
26	INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
27 28	<pre><type> BUF(*)</type></pre>
29	
30	MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
31	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
32	<type> BUF(*)</type>
33 34	MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
35	INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
36	<type> BUF(*)</type>
37	MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
38	INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
39	<type> BUF(*)</type>
40 41	MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
42	INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
43	<type> BUF(*)</type>
44	MPI_FILE_IWRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
45	INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
46	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
47 48	<type> BUF(*)</type>
-	

<pre>MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET <type> BUF(*)</type></pre>	1 2 3 4
<pre>MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR <type> BUF(*)</type></pre>	5 6 7 8
<pre>MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR <type> BUF(*)</type></pre>	9 10 11 12
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR) INTEGER COMM, AMODE, INFO, FH, IERROR CHARACTER*(*) FILENAME	12 13 14 15
MPI_FILE_PREALLOCATE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE	16 17 18
<pre>MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) INTEGER FH, COUNT, DATATYPE, IERROR <type> BUF(*)</type></pre>	19 20 21 22
<pre>MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type></pre>	23 24 25 26
<pre>MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type></pre>	27 28 29
<pre>MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR) INTEGER FH, COUNT, DATATYPE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET <type> BUF(*)</type></pre>	30 31 32 33 34
<pre>MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type></pre>	35 36 37
<pre>MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET <type> BUF(*)</type></pre>	38 39 40 41 42
<pre>MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET <type> BUF(*)</type></pre>	43 44 45 46
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	47 48

1 2	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type>
3 4 5 6	MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) INTEGER FH, COUNT, DATATYPE, IERROR <type> BUF(*)</type>
7 8 9 10	<pre>MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type></pre>
10 11 12 13	<pre>MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type></pre>
14 15 16 17	<pre>MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type></pre>
18 19 20	MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR) INTEGER FH, WHENCE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
21 22 23 24	MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR) INTEGER FH, WHENCE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
25 26 27	MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR LOGICAL FLAG
28 29 30	MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR
31 32 33	MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE
34 35 36 37 38	<pre>MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER(KIND=MPI_OFFSET_KIND) DISP CHARACTER*(*) DATAREP</pre>
39 40	MPI_FILE_SYNC(FH, IERROR) INTEGER FH, IERROR
41 42 43 44	<pre>MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) INTEGER FH, COUNT, DATATYPE, IERROR <type> BUF(*)</type></pre>
45 46 47 48	<pre>MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type></pre>

MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	1 2
<type> BUF(*)</type>	3
MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)	4
INTEGER FH, COUNT, DATATYPE, IERROR	5 6
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	7
<type> BUF(*)</type>	8
	9
MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR	10
<type> BUF(*)</type>	11
MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)	12 13
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	13
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	15
<type> BUF(*)</type>	16
MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)	17
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	18
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	19
<type> BUF(*)</type>	20 21
MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	21
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	23
<type> BUF(*)</type>	24
MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)	25
INTEGER FH, COUNT, DATATYPE, IERROR	26
<type> BUF(*)</type>	27
MOT FILE UNITE ODDEDED END (FIL DUE OTATUS TEDDOD)	28
MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR	29
<pre><type> BUF(*)</type></pre>	30
(cype> bor(*)	31 32
MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	32
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	34
<type> BUF(*)</type>	35
MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	36
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	37
<type> BUF(*)</type>	38
MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,	39
DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)	40
CHARACTER*(*) DATAREP	41
EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN	42
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	43
INTEGER IERROR	44
	45
	46

```
1
     A.5.13 Language Bindings Fortran Bindings
\mathbf{2}
     MPI_F_SYNC_REG(BUF)
3
         <type> BUF(*)
4
5
     MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
6
         INTEGER P, R, NEWTYPE, IERROR
7
     MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
8
         INTEGER R, NEWTYPE, IERROR
9
10
     MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
11
         INTEGER P, R, NEWTYPE, IERROR
12
     MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)
13
         INTEGER TYPECLASS, SIZE, DATATYPE, IERROR
14
15
     The following procedure is not available with mpif.h:
16
     MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
17
         TYPE(MPI_Status) :: F08_STATUS
18
         INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
19
     The following procedure is not available with mpif.h:
20
     MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
21
         INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
22
         TYPE(MPI_Status) :: F08_STATUS
23
24
25
     A.5.14 Tools / Profiling Interface Fortran Bindings
26
27
     MPI_PCONTROL(LEVEL)
         INTEGER LEVEL
28
29
30
             Deprecated Fortran Bindings
     A.5.15
^{31}
32
     MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
33
         INTEGER COMM, KEYVAL, IERROR
34
     MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
35
         INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
36
         LOGICAL FLAG
37
38
     MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
39
         INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
40
     MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
41
                   ATTRIBUTE_VAL_OUT, FLAG, IERR)
42
         INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
43
                    ATTRIBUTE_VAL_OUT, IERR
44
         LOGICAL FLAG
45
46
     MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
47
         INTEGER INFO, VALUELEN, IERROR
48
```

CHARACTER*(*) KEY, VALUE LOGICAL FLAG MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR) INTEGER INFO, VALUELEN, IERROR CHARACTER*(*) KEY LOGICAL FLAG MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR) EXTERNAL COPY_FN, DELETE_FN INTEGER KEYVAL, EXTRA_STATE, IERROR MPI_KEYVAL_FREE(KEYVAL, IERROR) INTEGER KEYVAL, IERROR MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR) INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, IERR LOGICAL FLAG MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR MPI_SIZEOF(X, SIZE, IERROR) <type> X INTEGER SIZE, IERROR

 24



Annex B

Change-Log

Annex B.1 summarizes changes from the previous version of the MPI standard to the version presented by this document. Only significant changes (i.e., clarifications and new features) that might either require implementation effort in the MPI libraries or change the understanding of MPI from a user's perspective are presented. Editorial modifications, formatting, typo corrections and minor clarifications are not shown. If not otherwise noted, the section and page references refer to the locations of the change or new functionality in this version of the standard. Changes in Annexes B.2–B.6 were already introduced in the corresponding sections in previous versions of this standard.

B.1 Changes from Vers	sion 3.2 to Version 4.0	23 24
B.1.1 Fixes to Errata in Pre		25
D.I.I TIXES to Ellata III TIE		26
	83, and MPI-3.1 Section 17.2.5 on page 657 line 11.	27
	STATUS_F2F08 and MPI_STATUS_F082F routines and	
declaration for TYPE(MP	Pl_Status) are not supposed to appear with mpif.h.	29
2. Sections 2.5.4, 19.2.5, an	nd A.1.1 on pages 18, 782, and 798, and MPI-3.1 Sect	tions 30
2.5.4, 17.2.5, and A.1.1 c	on pages 15, 656, and 669.	32
	MPI_F_STATUS_SIZE, MPI_F_SOURCE, MPI_F_TAG, and	33
MPI_F_ERROR.		34
3 Section 19.2.5 on page 7	84, and MPI-3.1 Section 17.2.5 on page 658.	35
	D IN parameters for MPI_STATUS_F2F08 and	36
MPI_STATUS_F082F.	The parameters for the 1_5 for 05_1 21 00 and	37
		38
B.1.2 Changes in MPI-4.0		39
		40
	7.3, 3.8.1, 3.8.2, 6.13, 14.4.5, and Annex A.2 on pages 11	, 44 , ⁴¹
56,62,76,79,246,630,50		42
The semantic terms were	e updated.	43
2. Section 16.3 on page 725		44
MPI_SIZEOF was deprec		45
WIT_SIZEOF was depree	ateu.	46
3. Chapter 4 on page 93 .		47
	ioned communication was added.	48
Unoffi	cial Draft for Comment Only	941

1 2 3	4.	Section 7.4.2 on page 291. MPI_COMM_TYPE_HW_UNGUIDED was added as a new possible value for the split_type parameter of the MPI_COMM_SPLIT_TYPE function.
4 5 6	5.	Section 15.3.8. A callback-driven event interface was added to the MPI tool information interface.
7 8 9 10 11 12	6.	Section 7.4.2 on page 291. MPI_COMM_TYPE_HW_GUIDED was added as a new possible value for the split_type parameter of the MPI_COMM_SPLIT_TYPE function, as well as a new info key "mpi_hw_resource_type". A specific value associated with this new info key is also defined: "mpi_shared_memory".
13 14 15 16 17	7.	Chapter 11, Sections 2.8, $3.2.3$, $7.2.4$, $7.3.2$, $7.4.2$, $7.6.2$, 8.4 , $8.5.1$, $8.5.3$, $8.5.4$, $9.1.1$, $9.1.2$, 9.3 , $9.3.4$, 9.5 , 11.6 , $14.2.1$, $14.2.7$, 14.7 , $15.3.4$, $19.2.4$, $19.2.6$, and Annex A on pages 443, 24, 31, 279, 282, 291, 322, 354, 356, 358, 360, 407, 409, 414, 422, 426, 472, 593, 599, 658, 674, 780, 785, and 795 The Sessions Model was added to the standard.
18 19 20	8.	Section 11.2.1 on page 444. A new function MPI_INFO_CREATE_ENV was added.
21 22 23	B.2	Changes from Version 3.1 to Version 3.2
24	B.2.1	Fixes to Errata in Previous Versions of MPI
25 26 27 28 29 30 31	1.	Sections 8.6.1, 8.6.2 and 8.9 on pages 380, 384 and 402, and MPI-3.1 Sections 7.6.1, 7.6.2 and 7.8 on pages 315, 318 and 329. MPI_NEIGHBOR_ALLTOALL{ $ V W$ } and MPI_NEIGHBOR_ALLGATHER{ $ V$ } for Cartesian virtual grids were clarified. An advice to implementors was added to illustrate a correct implementation for the case of periods[d]==1 or .TRUE. and dims[d]==1 or 2 in a direction d.
32 33	B.2.2	2 Changes in MPI-3.2
34 35 36	1.	Section 3.7 on page 54. The introduction of MPI nonblocking communication was changed to describe correctness and performance reasons for the use of nonblocking communication.
37 38 39 40	2.	Section 3.8.4 on page 84. Cancelling a send request by calling MPI_CANCEL has been deprecated and may be removed in a future version of the MPI specification.
41 42 43 44	3.	Sections 3.7.3, 3.9, 6.13, 8.8, and 8.9 on pages 62, 85, 246, 396, and 402. Persistent collective communication and persistent neighborhood communication were added to the standard.
45 46 47 48	4.	Section 7.4.2 on page 291, and MPI-3.1 Section 6.4.2 on page 237. The functions MPI_COMM_DUP and MPI_COMM_IDUP were updated to no longer propagate info hints. This change may affect backward compatibility.

5.	Sections 7.4.4, 12.2.7, and 14.2.8 on pages 308, 520, and 601, and MPI-3.1 Sections 6.4.4, 11.2.7, and 13.2.8 on pages 248, 415, and 500. The definition of info hints was updated to allow applications to provide assertions regarding their usage of MPI objects and operations.	1 2 3 4
6.	Section 7.4.4 on page 308. The new info hints "mpi_assert_no_any_tag", "mpi_assert_no_any_source", "mpi_assert_exact_length", and "mpi_assert_allow_overtaking" were added for use with communicators.	5 6 7 8 9
7.	Section 7.4.2 on page 291. The MPI_COMM_IDUP_WITH_INFO function was added.	10 11 12
8.	Sections 7.4.4, 12.2.7, and 14.2.8 on pages 308, 520, and 601. The semantics of the MPI_COMM_SET_INFO, MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, MPI_WIN_GET_INFO, MPI_FILE_SET_INFO, and MPI_FILE_GET_INFO were clarified.	13 14 15 16
9.	Section 15.3.10 and Table 15.7 on pages 718 and 720. MPI_T_ERR_INVALID_ITEM is deprecated. MPI routines should return MPI_T_ERR_INVALID_INDEX instead of MPI_T_ERR_INVALID_ITEM.	17 18 19 20
10.	Section 8.5. MPI_DIMS_CREATE is now guaranteed to return MPI_SUCCESS if the number of di- mensions passed to the routine is set to 0 and the number of nodes is set to 1.	21 22 23 24
11.	Sections 2.8, 9.3, 9.5, and 11.2.1 on pages 24, 414, 426, and 444. MPI calls that are not related to any objects are considered to be attached to the communicator MPI_COMM_SELF instead of MPI_COMM_WORLD. The definition of MPI_ERRORS_ARE_FATAL was clarified to cover all connected processes, and a new error handler, MPI_ERRORS_ABORT, was created to limit the scope of aborting.	25 26 27 28 29
12.	Sections 9.2, 12.2.2, and 12.2.3 on pages 411, 509, and 511. Introduced alignment requirements for memory allocated through MPI_ALLOC_MEM, MPI_WIN_ALLOCATE, and MPI_WIN_ALLOCATE_SHARED and added a new info key "mpi_minimum_memory_alignment" to specify a desired alternative minimum alignment.	30 31 32 33 34
13.	Section 9.4 on page 427. The error class MPI_ERR_PROC_ABORTED has been added.	35 36 37
14.	Section 13.3 on page 588 The mpi_f08 binding incorrectly had the dummy parameter flag in the MPI F08 binding for MPI_STATUS_SET_CANCELLED marked as INTENT(OUT). It has been fixed to be INTENT(IN).	38 39 40 41
15.	Sections 9.3 and 9.4 on pages 414 and 425. Clarified definition of errors to say that MPI should continue whenever possible and allow the user to recover from errors.	42 43 44 45
16.	Section 10. Added a new function MPI_INFO_GET_STRING that takes a buffer length argument for returning info value strings. This function returns the required buffer length for	46 47 48

1 2		the requested string and guarantees null termination for C strings where buffer size is greater than 0.
3 4 5	17.	Section 10 on page 435 and Section 16.4 on page 724. MPI_INFO_GET and MPI_INFO_GET_VALUELEN were deprecated.
6 7 8 9	18.	Section 9.4 on page 425. Added text to clarify what is implied about the status of MPI and user visible buffers when MPI functions return MPI_SUCCESS or other error codes.
10 11 12	19.	Section 11.2.1 on page 444. Section 11.10.4 on page 500. Clarified the semantic of failure and error reporting before (and during) MPI_INIT and after MPI_FINALIZE.
13 14 15 16 17	20.	Section 11.8.4 on page 485. Section 11.8.4 on page 485. Added the "mpi_initial_errhandler" reserved info key with the reserved values "mpi_errors_abort", "mpi_errors_are_fatal", and "mpi_errors_return" to the launch keys in MPI_COMM_SPAWN, MPI_COMM_SPAWN_MULTIPLE, and mpiexec
18 19 20	21.	Section 3.9 and 3.7 on pages 85 and 56. Addition of MPI_ISENDRECV and MPI_ISENDRECV_REPLACE.
21 22	B.3	Changes from Version 3.0 to Version 3.1
23 24	B.3.1	Fixes to Errata in Previous Versions of MPI
25 26 27 28	1.	Chapters 3–19, Annex A.4 on page 846, and Example 6.21 on page 217, and MPI-3.0 Chapters 3–17, Annex A.3 on page 707, and Example 5.21 on page 187. Within the mpi_f08 Fortran support method, BIND(C) was removed from all SUBROUTINE, FUNCTION, and ABSTRACT INTERFACE definitions.
29 30 31 32	2.	Section 3.2.5 on page 34, and MPI-3.0 Section 3.2.5 on page 30. The three public fields MPI_SOURCE, MPI_TAG, and MPI_ERROR of the Fortran derived type TYPE(MPI_Status) must be of type INTEGER.
33 34 35 36	3.	Section 3.8.2 on page 79, and MPI-3.0 Section 3.8.2 on page 67. The flag arguments of the Fortran interfaces of MPI_IMPROBE were originally incorrectly defined as INTEGER (instead as LOGICAL).
37 38 39	4.	Section 7.4.2 on page 291, and MPI-3.0 Section 6.4.2 on page 237. In the mpi_f08 binding of MPI_COMM_IDUP, the output argument newcomm is declared as ASYNCHRONOUS.
40 41 42 43	5.	Section 7.4.4 on page 308, and MPI-3.0 Section 6.4.4 on page 248. In the mpi_f08 binding of MPI_COMM_SET_INFO, the intent of comm is IN, and the optional output argument ierror was missing.
44 45 46 47 48	6.	Section 8.6 on page 379, and MPI-3.0 Sections 7.6, on pages 314. In the case of virtual general graph topolgies (created with MPI_CART_CREATE), the use of neighborhood collective communication is restricted to adjacency matrices with the number of edges between any two processes is defined to be the same for both processes (i.e., with a symmetric adjacency matrix).

7.	Section 9.1.1 on page 407, and MPI-3.0 Section 8.1.1 on page 335. In the mpi_f08 binding of MPI_GET_LIBRARY_VERSION, a typo in the resultlen argument was corrected.	1 2 3
8.	Sections 9.2 (MPI_ALLOC_MEM and MPI_ALLOC_MEM_CPTR), 12.2.2 (MPI_WIN_ALLOCATE and MPI_WIN_ALLOCATE_CPTR), 12.2.3 (MPI_WIN_ALLOCATE_SHARED and MPI_WIN_ALLOCATE_SHARED_CPTR), 12.2.3 (MPI_WIN_SHARED_QUERY and MPI_WIN_SHARED_QUERY_CPTR), 15.2.1 and 15.2.7 (Profiling interface), and corresponding sections in MPI-3.0. The linker name concept was substituted by defining specific procedure names.	4 5 6 7 8 9 10
9.	Section 12.2.1 on page 507, and MPI-3.0 Section 11.2.2 on page 407. The "same_size" info key can be used with all window flavors, and requires that all processes in the process group of the communicator have provided this info key with the same value.	11 12 13 14 15
10.	Section 12.3.4 on page 528, and MPI-3.0 Section 11.3.4 on page 424. Origin buffer arguments to MPI_GET_ACCUMULATE are ignored when the MPI_NO_OP operation is used.	16 17 18 19
11.	Section 12.3.4 on page 528, and MPI-3.0 Section 11.3.4 on page 424. Clarify the roles of origin, result, and target communication parameters in MPI_GET_ACCUMULATE.	19 20 21 22
12.	Section 15.3 on page 671, and MPI-3.0 Section 14.3 on page 561 New paragraph and advice to users clarifying intent of variable names in the tools information interface.	23 24 25 26
13.	Section 15.3.3 on page 673, and MPI-3.0 Section 14.3.3 on page 563. New paragraph clarifying variable name equivalence in the tools information interface.	27 28
14.	Sections 15.3.6, 15.3.7, and 15.3.9 on pages 678, 685, and 713, and MPI-3.0 Sections 14.3.6, 14.3.7, and 14.3.8 on pages 567, 573, and 584. In functions MPI_T_CVAR_GET_INFO, MPI_T_PVAR_GET_INFO, and MPI_T_CATEGORY_GET_INFO, clarification of parameters that must be identical for equivalent control variable / performance variable / category names across connected processes.	29 30 31 32 33 34 35
15.	Section 15.3.7 on page 685, and MPI-3.0 Section 14.3.7 on page 573. Clarify return code of MPI_T_PVAR_{START,STOP,RESET} routines.	36 37 38
16.	Section 15.3.7 on page 685, and MPI-3.0 Section 14.3.7 on page 579, line 7. Clarify the return code when bad handle is passed to an MPI_T_PVAR_* routine.	39 40
17.	Section 19.1.4 on page 737, and MPI-3.0 Section 17.1.4 on page 603. The advice to implementors at the end of the section was rewritten and moved into the following section.	41 42 43 44
18.	Section 19.1.5 on page 738, and MPI-3.0 Section 17.1.5 on page 605. The section was fully rewritten. The linker name concept was substituted by defining specific procedure names.	45 46 47 48

1 2	19.	Section 19.1.6 on page 743, and MPI-3.0 Section 17.1.6 on page 611. The requirements on BIND(C) procedure interfaces were removed.
3 4 5 6 7	20.	Annexes A.3, A.4, and A.5 on pages 819, 846, and 904, and MPI-3.0 Annexes A.2, A.3, and A.4 on pages 685, 707, and 756. The predefined callback MPI_CONVERSION_FN_NULL was added to all three annexes.
8 9 10 11 12	21.	Annex A.4.5 on page 868, and MPI-3.0 Annex A.3.4 on page 724. In the mpi_f08 binding of MPI_{COMM TYPE WIN}_{DUP NULL_COPY NULL_DELETE}_FN, all INTENT() information was removed.
13 14	B.3.2	Changes in MPI-3.1
15 16 17	1.	Sections 2.6.4 and 5.1.5 on pages 23 and 128. The use of the intrinsic operators "+" and "-" for absolute addresses is substituted by MPI_AINT_ADD and MPI_AINT_DIFF. In C, they can be implemented as macros.
18 19 20 21 22 23 24	2.	Sections 9.1.1, 11.2.1, and 11.6 on pages 407, 444, and 472. The routines MPI_INITIALIZED, MPI_FINALIZED, MPI_QUERY_THREAD, MPI_IS_THREAD_MAIN, MPI_GET_VERSION, and MPI_GET_LIBRARY_VERSION are callable from threads without restriction (in the sense of MPI_THREAD_MULTIPLE), irrespective of the actual level of thread support provided, in the case where the im- plementation supports threads.
25 26 27	3.	Section 12.2.1 on page 507. The "same_disp_unit" info key was added for use in RMA window creation routines.
28 29 30	4.	Sections 14.4.2 and 14.4.3 on pages 610 and 615. Added MPI_FILE_IREAD_AT_ALL, MPI_FILE_IWRITE_AT_ALL, MPI_FILE_IREAD_ALL, and MPI_FILE_IWRITE_ALL
31 32 33 34	5.	Sections 15.3.6, 15.3.7, and 15.3.9 on pages 678, 685, and 713. Clarified that NULL parameters can be provided in MPI_T_{CVAR PVAR CATEGORY}_GET_INFO routines.
35 36 37 38 39 40	6.	Sections 15.3.6, 15.3.7, 15.3.9, and 15.3.10 on pages 678, 685, 713, and 718. New routines MPI_T_CVAR_GET_INDEX, MPI_T_PVAR_GET_INDEX, MPI_T_CATEGORY_GET_INDEX, were added to support retrieving indices of vari- ables and categories. The error codes MPI_T_ERR_INVALID and MPI_T_ERR_INVALID_NAME were added to indicate invalid uses of the interface.
41 42	B.4	Changes from Version 2.2 to Version 3.0
43 44	B.4.1	Fixes to Errata in Previous Versions of MPI
45 46 47 48	1.	Sections 2.6.2 and 2.6.3 on pages 21 and 23, and MPI-2.2 Section 2.6.2 on page 17, lines $41-42$, Section 2.6.3 on page 18, lines $15-16$, and Section 2.6.4 on page 18, lines $40-41$.

This is an MPI-2 erratum: The scope for the reserved prefix MPI_{-} and the C++ namespace MPI is now any name as originally intended in MPI-1.

- 2. Sections 3.2.2, 6.9.2, 14.5.2 Table 14.2, and Annex A.1.1 on pages 29, 206, 642, and 795, and MPI-2.2 Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, 16.1.16 Table 16.1, and Annex A.1.1 on pages 27, 164, 433, 472 and 513 This is an MPI-2.2 erratum: New named predefined datatypes MPI_CXX_BOOL, MPI_CXX_FLOAT_COMPLEX, MPI_CXX_DOUBLE_COMPLEX, and MPI_CXX_LONG_DOUBLE_COMPLEX were added in C and Fortran corresponding to the C++ types bool, std::complex<float>, std::complex<double>, and 10 std::complex<long double>. These datatypes also correspond to the deprecated 11 C++ predefined datatypes MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEX, and 12MPI::LONG_DOUBLE_COMPLEX, which were removed in MPI-3.0. The non-standard 13 C++ types Complex<...> were substituted by the standard types 14std::complex<...>. 1516
- 3. Sections 6.9.2 on pages 206 and MPI-2.2 Section 5.9.2, page 165, line 47. This is an MPI-2.2 erratum: MPI_C_COMPLEX was added to the "Complex" reduction group.
- 4. Section 8.5.5 on page 367, and MPI-2.2, Section 7.5.5 on page 257, C++ interface on page 264, line 3. This is an MPI-2.2 erratum: The argument rank was removed and in/outdegree are now defined as int & indegree and int & outdegree in the C++ interface of MPI_DIST_GRAPH_NEIGHBORS_COUNT.
- 5. Section 14.5.2, Table 14.2 on page 642, and MPI-2.2, Section 13.5.3, Table 13.2 on page 433. This was an MPI-2.2 erratum: The MPI_C_BOOL "external32" representation is cor-

rected to a 1-byte size.

- 6. MPI-2.2 Section 16.1.16 on page 471, line 45. This is an MPI-2.2 erratum: The constant MPI::_LONG_LONG should be MPI::LONG_LONG.
- 7. Annex A.1.1 on page 795, Table "Optional datatypes (Fortran)," and MPI-2.2, Annex A.1.1, Table on page 517, lines 34, and 37–41. This is an MPI-2.2 erratum: The C++ datatype handles MPI::INTEGER16, MPI::REAL16, MPI::F_COMPLEX4, MPI::F_COMPLEX8, MPI::F_COMPLEX16, MPI::F_COMPLEX32 were added to the table.

Changes in MPI-3.0 B.4.2

- 1. Section 2.6.1 on page 21, Section 17.2 on page 728 and all other chapters. The C++ bindings were removed from the standard. See errata in Section B.4.1 on page 946 for the latest changes to the MPI C++ binding defined in MPI-2.2. This change may affect backward compatibility.
- 2. Section 2.6.1 on page 21, Section 16.1 on page 721 and Section 17.1 on page 727. The deprecated functions MPI_TYPE_HVECTOR, MPI_TYPE_HINDEXED, MPI_TYPE_STRUCT, MPI_ADDRESS, MPI_TYPE_EXTENT, MPI_TYPE_LB,

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

7

8

9

17

18

19

20

21

22

23

 24 25

26

27

28

2930

31

32

33

34

35

36

37

38 39

40 41

42

43

44

4546

47

1 2 3 4 5 6 7	MPI_TYPE_UB, MPI_ERRHANDLER_CREATE (and its callback function prototype MPI_Handler_function), MPI_ERRHANDLER_SET, MPI_ERRHANDLER_GET, the deprecated special datatype handles MPI_LB, MPI_UB, and the constants MPI_COMBINER_HINDEXED_INTEGER, MPI_COMBINER_HVECTOR_INTEGER, MPI_COMBINER_STRUCT_INTEGER were removed from the standard. This change may affect backward compatibility.
	Section 2.3 on page 10. Clarified parameter usage for IN parameters. C bindings are now const-correct where backward compatibility is preserved.
11 4. 12 13 14 15	Section 2.5.4 on page 18 and Section 8.5.4 on page 360. The recommended C implementation value for MPI_UNWEIGHTED changed from NULL to non-NULL. An additional weight array constant (MPI_WEIGHTS_EMPTY) was introduced.
17 18	Section 2.5.4 on page 18 and Section 9.1.1 on page 407. Added the new routine MPI_GET_LIBRARY_VERSION to query library specific versions, and the new constant MPI_MAX_LIBRARY_VERSION_STRING.
19 20 6. 21 22 23 24 25 26 27 28 29	Sections 2.5.8, 3.2.2, 3.3, 6.9.2, on pages 20, 29, 31, 206, Sections 5.1, 5.1.7, 5.1.8, 5.1.11, 13.3 on pages 109, 134, 136, 139, 588, and Annex A.1.1 on page 795. New inquiry functions, MPI_TYPE_SIZE_X, MPI_TYPE_GET_EXTENT_X, MPI_TYPE_GET_TRUE_EXTENT_X, and MPI_GET_ELEMENTS_X, return their results as an MPI_Count value, which is a new type large enough to represent element counts in memory, file views, etc. A new function, MPI_STATUS_SET_ELEMENTS_X, modifies the opaque part of an MPI_Status object so that a call to MPI_GET_ELEMENTS_X returns the provided MPI_Count value (in Fortran, INTEGER (KIND=MPI_COUNT_KIND)). The corresponding predefined datatype is MPI_COUNT.
31 32	Chapter 3 on page 27 through Chapter 19 on page 731. In the C language bindings, the array-arguments' interfaces were modified to consistently use use [] instead of *.
33 34 35 36	Exceptions are MPI_INIT, which continues to use char ***argv (correct because of subtle rules regarding the use of the & operator with char *argv []), and MPI_INIT_THREAD, which is changed to be consistent with MPI_INIT.
37 8. 38 39 40 41 42 43 44	Sections 3.2.5, 5.1.5, 5.1.11, 5.2 on pages 34, 128, 139, 159. The functions MPI_GET_COUNT and MPI_GET_ELEMENTS were defined to set the count argument to MPI_UNDEFINED when that argument would overflow. The functions MPI_PACK_SIZE and MPI_TYPE_SIZE were defined to set the size argument to MPI_UNDEFINED when that argument would overflow. In all other MPI-2.2 routines, the type and semantics of the count arguments remain unchanged, i.e., int or INTEGER.
	Section 3.2.6 on page 37, and Section 3.8 on page 76. MPI_STATUS_IGNORE can be also used in MPI_IPROBE, MPI_PROBE, MPI_IMPROBE, and MPI_MPROBE.

- Section 3.8 on page 76 and Section 3.10 on page 91. The use of MPI_PROC_NULL in probe operations was clarified. A special predefined message MPI_MESSAGE_NO_PROC was defined for the use of matching probe (i.e., the new MPI_MPROBE and MPI_IMPROBE) with MPI_PROC_NULL.
- 11. Sections 3.8.2, 3.8.3, 19.2.4, A.1.1 on pages 79, 81, 780, 795.

Like MPI_PROBE and MPI_IPROBE, the new MPI_MPROBE and MPI_IMPROBE operations allow incoming messages to be queried without actually receiving them, except that MPI_MPROBE and MPI_IMPROBE provide a mechanism to receive the specific message with the new routines MPI_MRECV and MPI_IMRECV regardless of other intervening probe or receive operations. The opaque object MPI_Message, the null handle MPI_MESSAGE_NULL, and the conversion functions MPI_Message_c2f and MPI_Message_f2c were defined.

- 12. Section 5.1.2 on page 111 and Section 5.1.13 on page 144. The routine MPI_TYPE_CREATE_HINDEXED_BLOCK and constant MPI_COMBINER_HINDEXED_BLOCK were added.
- Chapter 6 on page 171 and Section 6.12 on page 228.
 Added nonblocking interfaces to all collective operations.

14. Sections 7.4.2, 7.4.4, 12.2.7, on pages 291, 308, 520. The new routines MPI_COMM_DUP_WITH_INFO, MPI_COMM_SET_INFO, MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, and MPI_WIN_GET_INFO were added. The routine MPI_COMM_DUP must also duplicate info hints.

```
15. Section 7.4.2 on page 291.
Added MPI_COMM_IDUP.
```

16. Section 7.4.2 on page 291. Added the new communicator construction routine MPI_COMM_CREATE_GROUP, which is invoked only by the processes in the group of the new communicator being constructed.

- 17. Section 7.4.2 on page 291. Added the MPI_COMM_SPLIT_TYPE routine and the communicator split type constant MPI_COMM_TYPE_SHARED.
- 18. Section 7.6.2 on page 322. In MPI-2.2, communication involved in an MPI_INTERCOMM_CREATE operation could interfere with point-to-point communication on the parent communicator with the same tag or MPI_ANY_TAG. This interference has been removed in MPI-3.0.
- Section 7.8 on page 345.
 Section 6.8 on page 238. The constant MPI_MAX_OBJECT_NAME also applies for type and window names.
- 20. Section 8.5.8 on page 378. MPI_CART_MAP can also be used for a zero-dimensional topologies.

 $\mathbf{2}$

 $\overline{7}$

1	21.	Section 8.6 on page 379 and Section 8.7 on page 391.
2		The following neighborhood collective communication routines were added to sup-
3		port sparse communication on virtual topology grids: MPI_NEIGHBOR_ALLGATHER,
4		MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL,
5		MPI_NEIGHBOR_ALLTOALLV, MPI_NEIGHBOR_ALLTOALLW and the nonblocking
6		variants MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV,
7		MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and
8		
9		MPI_INEIGHBOR_ALLTOALLW. The displacement arguments in
		MPI_NEIGHBOR_ALLTOALLW and MPI_INEIGHBOR_ALLTOALLW were defined as
10		address size integers. In MPI_DIST_GRAPH_NEIGHBORS, an ordering rule was added
11		for communicators created with MPI_DIST_GRAPH_CREATE_ADJACENT.
12	<u> </u>	Section 11.2.1 on page 444 and Section 11.2.1 on page 447.
13	22.	The use of MPI_INIT, MPI_INIT_THREAD and MPI_FINALIZE was clarified. After
14		
15		MPI is initialized, the application can access information about the execution envi-
16		ronment by querying the new predefined info object MPI_INFO_ENV.
17	23	Section 11.2.1 on page 444.
18	20.	Allow calls to MPI_T routines before MPI_INIT and after MPI_FINALIZE.
19		Anow cans to write ridulines before write write write write ridule.
20	24.	Chapter 12 on page 505.
21		Substantial revision of the entire One-sided chapter, with new routines for window
22		creation, additional synchronization methods in passive target communication, new
23		one-sided communication routines, a new memory model, and other changes.
24		ono sidod communicación rodonico, a non monory modon and ochor changes.
25	25.	Section 15.3 on page 671.
26		A new MPI Tool Information Interface was added.
27		The following changes are related to the Fortran language support.
28		The following changes are related to the Portrain language support.
29	26.	Section 2.3 on page 10, and Sections 19.1.1, 19.1.2, 19.1.7 on pages 731, 732, and 747.
30		The new mpi_08 Fortran module was introduced.
31		
	27.	Section 2.5.1 on page 16, and Sections 19.1.2, 19.1.3, 19.1.7 on pages 732, 735, and 747.
32		Handles to opaque objects were defined as named types within the mpi_08 Fortran
33		module. The operators .EQ., .NE., ==, and /= were overloaded to allow the compari-
34		son of these handles. The handle types and the overloaded operators are also available
35		through the mpi Fortran module.
36		
37	28.	Sections 2.5.4, 2.5.5 on pages 18, 19, Sections 19.1.1, 19.1.10, 19.1.11, 19.1.12, 19.1.13
38		on pages 731, 757, 759, 759, 762, and Sections 19.1.2, 19.1.3, 19.1.7 on pages 732, 735,
39		747.
40		Within the mpi_08 Fortran module, choice buffers were defined as assumed-type and
41		assumed-rank according to Fortran 2008 TS 29113 $[46]$, and the compile-time constant
42		MPI_SUBARRAYS_SUPPORTED was set to .TRUE With this, Fortran subscript triplets
43		can be used in nonblocking MPI operations; vector subscripts are not supported in
44		nonblocking operations. If the compiler does not support this Fortran TS 29113
45		feature, the constant is set to .FALSE
46		
47	29.	Section 2.6.2 on page 21, Section 19.1.2 on page 732, and Section 19.1.7 on page 747.
48		The ierror dummy arguments are OPTIONAL within the mpi_08 Fortran module.

30. Section 3.2.5 on page 34, Sections 19.1.2, 19.1.3, 19.1.7, on pages 732, 735, 747, and Section 19.2.5 on page 782. Within the mpi_08 Fortran module, the status was defined as TYPE(MPI_Status). Additionally, within both the mpi and the mpi_f08 modules, the constants MPI_STATUS_SIZE, MPI_SOURCE, MPI_TAG, MPI_ERROR, and TYPE(MPI_Status) are defined. New conversion routines were added: MPI_STATUS_F2F08, MPI_STATUS_F082F, MPI_Status_c2f08, and MPI_Status_f082c, In mpi.h, the new type MPI_F08_status, and the external variables MPI_F08_STATUS_IGNORE and MPI_F08_STATUSES_IGNORE were added.

31. Section 3.6 on page 52. In Fortran with the mpi module or mpif.h, the type of the buffer_addr argument of MPI_BUFFER_DETACH is incorrectly defined and the argument is therefore unused.

- 32. Section 5.1 on page 109, Section 5.1.6 on page 132, and Section 19.1.15 on page 763. The Fortran alignments of basic datatypes within Fortran derived types are implementation dependent; therefore it is recommended to use the BIND(C) attribute for derived types in MPI communication buffers. If an array of structures (in C/C++) or derived types (in Fortran) is to be used in MPI communication buffers, it is recommended that the user creates a portable datatype handle and additionally applies MPI_TYPE_CREATE_RESIZED to this datatype handle.
- 33. Sections 5.1.10, 6.9.5, 6.9.7, 7.7.4, 7.8, 9.3.1, 9.3.2, 9.3.3, 16.1, 19.1.9 on pages 139, 22213, 219, 339, 345, 417, 419, 421, 721, and 749. In some routines, the dummy ar-23gument names were changed because they were identical to the Fortran keywords TYPE and FUNCTION. The new dummy argument names must be used because the mpi and mpi 08 modules guarantee keyword-based actual argument lists. The argument name type was changed in MPI_TYPE_DUP, the Fortran USER_FUNCTION of MPI_OP_CREATE, MPI_TYPE_SET_ATTR, MPI_TYPE_GET_ATTR, MPI_TYPE_DELETE_ATTR, MPI_TYPE_SET_NAME, 29 MPI_TYPE_GET_NAME, MPI_TYPE_MATCH_SIZE, the callback prototype definition MPI_Type_delete_attr_function, and the predefined callback function MPI_TYPE_NULL_DELETE_FN; function was changed in MPI_OP_CREATE, MPI_COMM_CREATE_ERRHANDLER, MPI_WIN_CREATE_ERRHANDLER, 34 MPI_FILE_CREATE_ERRHANDLER, and MPI_ERRHANDLER_CREATE. For consistency reasons, INOUBUF was changed to INOUTBUF in MPI_REDUCE_LOCAL, and intracomm to newintracomm in MPI_INTERCOMM_MERGE.

34.	Section $7.7.2$ on page 330 .	
	It was clarified that in Fortran, the flag values returned by a comm_copy_attr_fn	
	callback, including MPI_COMM_NULL_COPY_FN and MPI_COMM_DUP_FN, are	
	.FALSE. and .TRUE.; see MPI_COMM_CREATE_KEYVAL.	
35	Section 9.2 on page 411.	

With the mpi and mpi_f08 Fortran modules, MPI_ALLOC_MEM now also supports TYPE(C_PTR) C-pointers instead of only returning an address-sized integer that may be usable together with a non-standard Cray-pointer.

36. Section 19.1.15 on page 763, and Section 19.1.7 on page 747. 4748 Fortran SEQUENCE and BIND(C) derived application types can now be used as buffers

1

2

3

4

5

6 $\overline{7}$

8

9 10

11

12

13 14

15

16

17

18

19

20

21

 24 25

26

27

28

30 31

32 33

35

43

44

45

1		in MPI operations.
2 3 4	37.	Section 19.1.16 on page 765 to Section 19.1.19 on page 774, Section 19.1.7 on page 747, and Section 19.1.8 on page 748.
5 6 7		The sections about Fortran optimization problems and their solutions were partially rewritten and new methods are added, e.g., the use of the ASYNCHRONOUS attribute. The constant MPI_ASYNC_PROTECTS_NONBLOCKING tells whether the semantics of
8 9 10		the ASYNCHRONOUS attribute is extended to protect nonblocking operations. The For- tran routine MPI_F_SYNC_REG is added. MPI-3.0 compliance for an MPI library together with a Fortran compiler is defined in Section 19.1.7.
11 12 13 14	38.	Section 19.1.2 on page 732. Within the mpi_08 Fortran module, dummy arguments are now declared with INTENT=IN, OUT, or INOUT as defined in the mpi_08 interfaces.
15 16	39.	Section 19.1.3 on page 735, and Section 19.1.7 on page 747. The existing mpi Fortran module must implement compile-time argument checking.
17 18 19	40.	Section 19.1.4 on page 737. The use of the mpif.h Fortran include file is now strongly discouraged.
20 21 22 23 24 25	41.	Section A.1.1, Table " <i>Predefined functions</i> " on page 803, Section A.1.3 on page 810, and Section A.4.5 on page 868. Within the new mpi_f08 module, all callback prototype definitions are now defined with explicit interfaces PROCEDURE(MPI) that have the BIND(C) attribute; user-written callbacks must be modified if the mpi_f08 module is used.
26 27 28 29	42.	Section A.1.3 on page 810. In some routines, the Fortran callback prototype names were changed from \ldots _FN to \ldots _FUNCTION to be consistent with the other language bindings.
30 31	B.5	Changes from Version 2.1 to Version 2.2
32 33	1.	Section 2.5.4 on page 18. It is now guaranteed that predefined named constant handles (as other constants)
34 35 36		can be used in initialization expressions or assignments, i.e., also before the call to MPI_INIT.
37 38 39	2.	Section 2.6 on page 21, and Section 17.2 on page 728. The C++ language bindings have been deprecated and may be removed in a future version of the MPI specification.
40 41 42 43 44 45 46	3.	Section 3.2.2 on page 29. MPI_CHAR for printable characters is now defined for C type char (instead of signed char). This change should not have any impact on applications nor on MPI libraries (except some comment lines), because printable characters could and can be stored in any of the C types char, signed char, and unsigned char, and MPI_CHAR is not allowed for predefined reduction operations.
47 48	4.	Section 3.2.2 on page 29. MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL,

1 MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and 2 MPI_C_LONG_DOUBLE_COMPLEX are now valid predefined MPI datatypes. 5. Section 3.4 on page 44, Section 3.7.2 on page 56, Section 3.9 on page 85, and Section 6.14 on page 171. 5 The read access restriction on the send buffer for blocking, non blocking and collective 6 API has been lifted. It is permitted to access for read the send buffer while the 7 operation is in progress. 9 6. Section 3.7 on page 54. 10 The Advice to users for IBSEND and IRSEND was slightly changed. 11 7. Section 3.7.3 on page 62. 12The advice to free an active request was removed in the Advice to users for 13 MPI_REQUEST_FREE. 14 158. Section 3.7.6 on page 75. 16MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input. 1718 9. Section 6.8 on page 198. 19 "In place" option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and 20MPI_ALLTOALLW for intracommunicators. 2110. Section 6.9.2 on page 206. 22 Predefined parameterized datatypes (e.g., returned by 23MPI_TYPE_CREATE_F90_REAL) and optional named predefined datatypes (e.g. 24 MPI_REAL8) have been added to the list of valid datatypes in reduction operations. 252611. Section 6.9.2 on page 206. 27 $MPI_(U)INT\{8,16,32,64\}_T$ are all considered C integer types for the purposes of the 28predefined reduction operators. MPI_AINT and MPI_OFFSET are considered Fortran 29 integer types. MPI_C_BOOL is considered a Logical type. 30 MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and 31MPI_C_LONG_DOUBLE_COMPLEX are considered Complex types. 32 33 12. Section 6.9.7 on page 219. 34 The local routines MPI_REDUCE_LOCAL and MPI_OP_COMMUTATIVE have been 35added. 36 13. Section 6.10.1 on page 221. 37 The collective function MPI_REDUCE_SCATTER_BLOCK is added to the MPI stan-38 dard. 39 40 14. Section 6.11.2 on page 225. 41 Added in place argument to MPI_EXSCAN. 4215. Section 7.4.2 on page 291, and Section 7.6 on page 318. 43 44Implementations that did not implement MPI_COMM_CREATE on intercommunicators will need to add that functionality. As the standard described the behav-45ior of this operation on intercommunicators, it is believed that most implementa-4647tions already provide this functionality. Note also that the C++ binding for both

Unofficial Draft for Comment Only

MPI_COMM_CREATE and MPI_COMM_SPLIT explicitly allow Intercomms.

1 2 3 4 5	16.	Section 7.4.2 on page 291. MPI_COMM_CREATE is extended to allow several disjoint subgroups as input if comm is an intracommunicator. If comm is an intercommunicator it was clarified that all processes in the same local group of comm must specify the same value for group.
6 7 8 9 10	17.	Section 8.5.4 on page 360. New functions for a scalable distributed graph topology interface has been added. In this section, the functions MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE, the constants MPI_UNWEIGHTED, and the derived C++ class Distgraphcomm were added.
11 12 13 14 15	18.	Section 8.5.5 on page 367. For the scalable distributed graph topology interface, the functions MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS and the constant MPI_DIST_GRAPH were added.
16 17 18	19.	Section 8.5.5 on page 367. Remove ambiguity regarding duplicated neighbors with MPI_GRAPH_NEIGHBORS and MPI_GRAPH_NEIGHBORS_COUNT.
19 20 21	20.	Section 9.1.1 on page 407. The subversion number changed from 1 to 2.
22 23 24 25	21.	Section 9.3 on page 414, Section 16.2 on page 724, and Annex A.1.3 on page 810. Changed function pointer typedef names MPI_{Comm,File,Win}_errhandler_fn to MPI_{Comm,File,Win}_errhandler_function. Deprecated old "_fn" names.
26 27 28 29	22.	Section 11.2.4 on page 454. Attribute deletion callbacks on MPI_COMM_SELF are now called in LIFO order. Implementors must now also register all implementation-internal attribute deletion callbacks on MPI_COMM_SELF before returning from MPI_INIT/MPI_INIT_THREAD.
30 31 32 33 34 35 36 37	23.	Section 12.3.4 on page 528. The restriction added in MPI 2.1 that the operation MPI_REPLACE in MPI_ACCUMULATE can be used only with predefined datatypes has been removed. MPI_REPLACE can now be used even with derived datatypes, as it was in MPI 2.0. Also, a clarification has been made that MPI_REPLACE can be used only in MPI_ACCUMULATE, not in collective operations that do reductions, such as MPI_REDUCE and others.
38 39 40 41 42	24.	Section 13.2 on page 581. Add "*" to the query_fn, free_fn, and cancel_fn arguments to the C++ binding for MPI::Grequest::Start() for consistency with the rest of MPI functions that take function pointer arguments.
43 44 45 46 47 48	25.	Section 14.5.2 on page 641, and Table 14.2 on page 642. MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, MPI_C_LONG_DOUBLE_COMPLEX, and MPI_C_BOOL are added as predefined datatypes in the external32 representation.

26.	Section 19.2.7 on page 787. The description was modified that it only describes how an MPI implementation behaves, but not how MPI stores attributes internally. The erroneous MPI-2.1 Example 16.17 was replaced with three new examples 19.13, 19.14, and 19.15 on pages 788–790 explicitly detailing cross-language attribute behavior. Implementations that matched the behavior of the old example will need to be updated.	1 2 3 4 5 6
27.	Annex A.1.1 on page 795. Removed type MPI::Fint (compare MPI_Fint in Section A.1.2 on page 809).	7 8 9
28.	Annex A.1.1 on page 795. Table Named Predefined Datatypes. Added MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL, MPI_C_FLOAT_COMPLEX, MPI_C_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are added as predefined datatypes.	10 11 12 13 14
B.6	Changes from Version 2.0 to Version 2.1	15 16
1.	Section 3.2.2 on page 29, and Annex A.1 on page 795. In addition, the MPI_LONG_LONG should be added as an optional type; it is a synonym for MPI_LONG_LONG_INT.	17 18 19 20
2.	Section 3.2.2 on page 29, and Annex A.1 on page 795. MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym), MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings.	21 22 23 24
3.	Section 3.2.5 on page 34. MPI_GET_COUNT with zero-length datatypes: The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI_UNDEFINED is returned.	25 26 27 28 29 30
4.	Section 5.1 on page 109. General rule about derived datatypes: Most datatype constructors have replication count or block length arguments. Allowed values are non-negative integers. If the value is zero, no elements are generated in the type map and there is no effect on datatype bounds or extent.	31 32 33 34 35 36
5.	Section 5.3 on page 166. MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.	37 38 39
6.	Section 6.9.6 on page 218. If comm is an intercommunicator in MPI_ALLREDUCE, then both groups should provide count and datatype arguments that specify the same type signature (i.e., it is not necessary that both groups provide the same count value).	40 41 42 43 44
7.	Section 7.3.1 on page 280. MPI_GROUP_TRANSLATE_RANKS and MPI_PROC_NULL: MPI_PROC_NULL is a valid rank for input to MPI_GROUP_TRANSLATE_RANKS, which returns MPI_PROC_NULL as the translated rank.	45 46 47 48

1 2	8.	Section 7.7 on page 328. About the attribute caching functions:
3 4 5 6 7 8 9 10		Advice to implementors. High-quality implementations should raise an error when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL is used with an object of the wrong type with a call to MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each keyval, information on the type of the associated user function. (<i>End of advice to implementors.</i>)
11 12 13 14 15 16	9.	Section 7.8 on page 345. In MPI_COMM_GET_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_OBJECT_NAME-1. In For- tran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_OBJECT_NAME.
17 18 19	10.	Section 8.4 on page 354. About MPI_GRAPH_CREATE and MPI_CART_CREATE: All input arguments must have identical values on all processes of the group of comm_old.
20 21 22 23 24	11.	Section 8.5.1 on page 356. In MPI_CART_CREATE: If ndims is zero then a zero-dimensional Cartesian topology is created. The call is erroneous if it specifies a grid that is larger than the group size or if ndims is negative.
25 26 27	12.	Section 8.5.3 on page 358. In MPI_GRAPH_CREATE: If the graph is empty, i.e., nnodes $== 0$, then MPI_COMM_NULL is returned in all processes.
28 29 30 31 32 33	13.	Section 8.5.3 on page 358. In MPI_GRAPH_CREATE: A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be non-symmetric.
34 35 36 37		Advice to users. Performance implications of using multiple edges or a non- symmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. (End of advice to users.)
38 39 40 41	14.	Section 8.5.5 on page 367. In MPI_CARTDIM_GET and MPI_CART_GET: If comm is associated with a zero- dimensional Cartesian topology, MPI_CARTDIM_GET returns ndims=0 and MPI_CART_GET will keep all output arguments unchanged.
42 43 44 45	15.	Section 8.5.5 on page 367. In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topol- ogy, coord is not significant and 0 is returned in rank.
46 47 48	16.	Section 8.5.5 on page 367. In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.

17.	Section 8.5.6 on page 375. In MPI_CART_SHIFT: It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology.	1 2 3 4 5
18.	Section 8.5.7 on page 377. In MPI_CART_SUB: If all entries in remain_dims are false or comm is already associ- ated with a zero-dimensional Cartesian topology then newcomm is associated with a zero-dimensional Cartesian topology.	6 7 8 9 10
18.1.	Section 9.1.1 on page 407. The subversion number changed from 0 to 1.	10 11 12 13
19.	Section 9.1.2 on page 409. In MPI_GET_PROCESSOR_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME.	14 15 16 17 18
20.	Section 9.3 on page 414. MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark the error handler for deallocation. This provides behavior similar to that of MPI_COMM_GROUP and MPI_GROUP_FREE.	19 20 21 22 23 24 25 26
21.	Section 11.2.1 on page 444, see explanations to MPI_FINALIZE. MPI_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI_COMM_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 11.10.4 on page 500.	27 28 29 30 31 32
22.	Section 11.2.1 on page 444. About MPI_ABORT:	33 34 35
	Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the MPI library but not mandatory. (<i>End of advice to users.</i>)	36 37 38
	Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. mpiexec or singleton init). (End of advice to implementors.)	39 40 41 42
23.	Section 10 on page 435. An implementation must support info objects as caches for arbitrary (key, value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an MPI_Info must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should	43 44 45 46 47 48

1 2 3		react if it recognizes a key but not the associated value. MPI_INFO_GET_NKEYS, MPI_INFO_GET_NTHKEY, MPI_INFO_GET_VALUELEN, and MPI_INFO_GET must retain all (key,value) pairs so that layered functionality can also use the Info object.
4 5 6 7 8	24.	Section 12.3 on page 522. MPI_PROC_NULL is a valid target rank in the MPI RMA calls MPI_ACCUMULATE, MPI_GET, and MPI_PUT. The effect is the same as for MPI_PROC_NULL in MPI point- to-point communication. See also item 25 in this list.
9 10 11 12 13	25.	Section 12.3 on page 522. After any RMA operation with rank MPI_PROC_NULL, it is still necessary to finish the RMA epoch with the synchronization method that started the epoch. See also item 24 in this list.
14 15 16	26.	Section 12.3.4 on page 528. MPI_REPLACE in MPI_ACCUMULATE, like the other predefined operations, is defined only for the predefined MPI datatypes.
17 18 19 20 21 22	27.	Section 14.2.8 on page 601. About MPI_FILE_SET_VIEW and MPI_FILE_SET_INFO: When an info object that specifies a subset of valid hints is passed to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.
23 24 25 26	28.	Section 14.2.8 on page 601. About MPI_FILE_GET_INFO: If no hint exists for the file associated with fh, a handle to a newly created info object is returned that contains no key/value pair.
27 28 29 30	29.	Section 14.3 on page 604. If a file does not have the mode MPI_MODE_SEQUENTIAL, then MPI_DISPLACEMENT_CURRENT is invalid as disp in MPI_FILE_SET_VIEW.
31 32	30.	Section 14.5.2 on page 641. The bias of 16 byte doubles was defined with 10383. The correct value is 16383.
33 34 35	31.	MPI-2.2, Section 16.1.4 (Section was removed in MPI-3.0). In the example in this section, the buffer should be declared as const void* buf.
36 37	32.	Section 19.1.9 on page 749. About MPI_TYPE_CREATE_F90_XXX:
38 39 40 41 42 43 44 45 46 47 48		Advice to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_XXX with the same combination of (XXX,p,r) . The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, the MPI implementation should return the same datatype handle for the same (REAL/COMPLEX/INTEGER,p,r) combination. Checking for the combination (p,r) in the preceding call to MPI_TYPE_CREATE_F90_XXX and using a hash- table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,p,r) . (End of advice to implementors.)

33.	Section A.1.1 on page 795.				
	MPI_BOTTOM is defined as void	*	const	MPI::BOTTOM.	

Unofficial Draf	t for	Comment	Only
-----------------	-------	---------	------



Bibliography

		10
[1]	Reverse domain name notation convention. https://docs.oracle.com/javase/ tutorial/java/package/namingpkgs.html. 7.4.2	11 12
[2]	V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and com- munication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2	13 14 15 16
[3]	V. Bala, S. Kipnis, L. Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2	17 18 19 20
[4]	Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In OON-SKI '94, page in press, 1994. 7.1	20 21 22
[5]	A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and debugging in a heterogeneous environment. <i>IEEE Computer</i> , 26(6):88–95, June 1993. 1.2	23 24 25 26
[6]	Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. <i>Parallel Computing</i> , 15:119–132, 1990. 1.2	27 28 29 30
[7]	Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. $IJHPCN$, $1(1/2/3)$:91–99, 2004. 12.7	31 32
[8]	Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In <i>Proceedings of Supercomputing '93</i> , pages 452–461, 1993. 14.1	33 34 35 36
[9]	R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report TM-ANL–92/17, Argonne National Laboratory, 1992. 1.2	37 38 20
[10]	Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. <i>Parallel Computing</i> , 20(4):547–564, April 1994. Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493. 1.2	39 40 41 42 43
[11]	Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the PARMACS message-passing library. <i>Parallel Computing</i> , 20(4):615–632, April 1994. 1.2	44 45 46 47 48

1 $\mathbf{2}$ 3 4 $\mathbf{5}$

1 2 3	[12]	S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormhole- routed networks for interprocessor communication in multicomputers. In <i>Proceedings</i> of the 1990 Supercomputing Conference, pages 647–656, 1990. 8.1
4 5 6 7	[13]	S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communication performance of large multicomputers. In <i>Proceedings of the 1991 International Conference on Parallel Processing, vol. II (Software)</i> , pages II–1–II–4, 1991. 8.1
8 9 10	[14]	Parasoft Corporation. Express version 1.0: A communication environment for parallel computers, 1988. 1.2
11 12 13 14	[15]	Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors. Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings, volume 6960 of Lecture Notes in Computer Science. Springer, 2011. 17, 40
15 16 17 18 19	[16]	Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In <i>IPPS '93 Workshop on Input/Output in Parallel Computer Systems</i> , pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38. 14.1
20 21 22	[17]	James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar Krishnan, Vinod Tipparaju, and Abhinav Vishnu. Noncollective communicator creation in MPI. In Cotronis et al. [15], pages 282–291. 7.4.2
23 24 25 26	[18]	J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. <i>Computers in Physics</i> , 7(2):166–75, April 1993. 1.2
27 28 29 30	[19]	J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993. 1.2
31 32	[20]	Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Concepts, June 1991. 1.2
33 34 35	[21]	Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Version 1.0 Interface, May 1992. 1.2
36 37 38 39	[22]	D. Feitelson. Communicators: Object-based multiparty interactions for parallel programming. Technical Report 91-12, Dept. Computer Science, The Hebrew University of Jerusalem, November 1991. 7.1.2
40 41 42	[23]	Message Passing Interface Forum. MPI: A Message-Passing Interface standard. The In- ternational Journal of Supercomputer Applications and High Performance Computing, 8, 1994. 1.3
43 44 45	[24]	Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report, 1995. http://www.mpi-forum.org. 1.3
46 47 48	[25]	Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. <i>PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network</i> <i>Parallel Computing.</i> MIT Press, 1994. 11.1

- [26] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A portable instrumented communications library, C reference manual. Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990. 1.2
- [27] Brice Goglin, Emmanuel Jeannot, Farouk Mansouri, and Guillaume Mercier. Hardware topology management in MPI applications through hierarchical communicators. *Parallel Computing*, 76:70–90, 2018. 7.4.2
- [28] Ryan E. Grant, Matthew G. F. Dosanjh, Michael J. Levenhagen, Ron Brightwell, and Anthony Skjellum. Finepoints: Partitioned multithreaded MPI communication. In ISC High Performance Conference (ISC), 2019. 4.1
- [29] Ryan E. Grant, Anthony Skjellum, and Purushotham V. Bangalore. Lightweight threading with MPI using persistent communications semantics. In Workshop on Exascale MPI (ExaMPI). Held in conjunction with the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC15), 2015. 4.1
- [30] D. Gregor, T. Hoefler, B. Barrett, and A. Lumsdaine. Fixing probe for multi-threaded MPI applications. Technical Report 674, Indiana University, Jan. 2009. 3.8.2
- [31] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993. 1.2
- [32] Michael Hennecke. A Fortran 90 interface to MPI version 1.1. Technical Report Internal Report 63/96, Rechenzentrum, Universität Karlsruhe, D-76128 Karlsruhe, Germany, June 1996. 19.1.3
- [33] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. de Supinski, and A. Lumsdaine. Efficient MPI support for advanced hybrid programming models. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 50–61. Springer, Sep. 2010. 3.8.1, 3.8.2
- [34] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, Sep. 2007. 6.12
- [35] T. Hoefler, F. Lorenzen, and A. Lumsdaine. Sparse non-blocking collectives in quantum mechanical calculations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 15th European PVM/MPI Users' Group Meeting, volume LNCS 5205, pages 55–63. Springer, Sep. 2008. 8.6
- [36] T. Hoefler and A. Lumsdaine. Message progression in parallel computing to thread or not to thread? In Proceedings of the 2008 IEEE International Conference on Cluster Computing. IEEE Computer Society, Oct. 2008. 6.12
- [37] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07.* IEEE Computer Society/ACM, Nov. 2007. 6.12

1 2

3

4

5

6

7 8

9

10

11 12

13

14

15 16

17

18 19

20

21

22

23

24 25

26

27

28

29 30

 31

32

33

34

35

36

37 38

39

40

 $41 \\ 42$

43

44

1 2 3 4	[38]	T. Hoefler, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication optimization for medical image reconstruction algorithms. In <i>Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting</i> , volume LNCS 5205, pages 75–83. Springer, Sep. 2008. 6.12
5 6 7 8	[39]	T. Hoefler and J. L. Träff. Sparse collective operations for MPI. In Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop, May 2009. 8.6
9 10 11	[40]	Torsten Hoefler and Marc Snir. Writing parallel libraries with MPI — common practice, issues, and extensions. In Cotronis et al. [15], pages 345–355. 7.4.2
12 13 14 15	[41]	Daniel J. Holmes, Bradley Morgan, Anthony Skjellum, Purushotham V. Bangalore, and Srinivas Sridharan. Planning for performance: Enhancing achievable performance for MPI through persistent collective operations. <i>Parallel Computing</i> , 81:32 – 57, 2019. 6.13
16 17 18	[42]	Institute of Electrical and Electronics Engineers, New York. <i>IEEE Standard for Binary Floating-Point Arithmetic, IEEE Standard 754-2008</i> , 2008. 14.5.2
19 20 21 22	[43]	International Organization for Standardization, Geneva. ISO 8859-1:1987: Informa- tion processing — 8-bit single-byte coded graphic character sets — Part 1: Latin al- phabet No. 1, February 1987. 14.5.2
23 24 25	[44]	International Organization for Standardization, Geneva. ISO/IEC 9945-1:1996: Infor- mation technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language], December 1996. 11.6, 14.2.1
26 27 28 29	[45]	International Organization for Standardization, Geneva. ISO/IEC 1539-1:2010: Information technology — Programming languages — Fortran — Part 1: Base language, November 2010. 19.1.1, 19.1.2
30 31 32 33	[46]	International Organization for Standardization, Geneva. <i>ISO/IEC TS 29113:2012:</i> Information technology — Further interoperability of Fortran with C, December 2012. 19.1.1, 19.1.1, 19.1.2, 19.1.7, 28
34 35	[47]	Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. <i>The High Performance Fortran Handbook</i> . MIT Press, 1993. 5.1.4
36 37 38 39	[48]	David Kotz. Disk-directed I/O for MIMD multiprocessors. In <i>Proceedings of the 1994</i> Symposium on Operating Systems Design and Implementation, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994. 14.1
40 41 42	[49]	O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. <i>Parallel Computing</i> , 9:213–225, 1989. 8.1
43 44 45 46 47 48	[50]	S. J. Lefflet, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, and C. Torek. An advanced 4.4BSD interprocess communication tutorial, Unix programmer's supplementary documents (PSD) 21. Technical report, Computer Systems Research Group, Depertment of Electrical Engineering and Computer Science, University of California, Berkeley, 1993. Available online: https://docs.freebsd.org/44doc/psd/21.ipc/paper.pdf. 11.10.5

[51]	Message Passing Interface Forum. Summary of the semantics of all operation-related MPI procedures, 2020. Available online: https://www.mpi-forum.org/docs. A.2	1 2
[52]	Bradley Morgan, Daniel J. Holmes, Anthony Skjellum, Purushotham Bangalore, and Srinivas Sridharan. Planning for performance: Persistent collective operations for MPI. In <i>Proceedings of the 24th European MPI Users' Group Meeting</i> , EuroMPI '17, pages 4:1–4:11, New York, NY, USA, 2017. ACM. 6.13	3 4 5 6 7
[53]	nCUBE Corporation. nCUBE 2 Programmers Guide, r2.0, December 1990. 1.2	8 9
[54]	Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992. 14.1	10 11
[55]	William J. Nitzberg. <i>Collective Parallel I/O</i> . PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995. 14.1	12 13 14
[56]	4.4BSD Programmer's Supplementary Documents (PSD). O'Reilly and Associates, 1994. 11.10.5	15 16
[57]	Paul Pierce. The NX/2 operating system. In Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, pages 384–390. ACM Press, 1988. 1.2	17 18 19 20
[58]	Martin Schulz and Bronis R. de Supinski. P^N MPI tools: A whole lot greater than the sum of their parts. In ACM/IEEE Supercomputing Conference (SC), pages 1–10. ACM, 2007. 15.2.8	21 22 23 24
[59]	K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In <i>Proceedings of Supercomputing '95</i> , December 1995. 14.1	25 26 27
[60]	A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, <i>Proceedings of the Fifth Distributed Memory Concurrent Computing Conference</i> , pages 767–776. IEEE Press, 1990. 1.2, 7.1.2	28 29 30 31
[61]	A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992. 1.2	32 33 34 35
[62]	Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, <i>Proceedings of the Scalable Parallel Libraries Conference</i> , pages 166–173. IEEE Computer Society Press, October 1993. 7.1	36 37 38 39 40
[63]	Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator ex- tensions to MPI in the MPIX (MPI eXtension) Library. Technical Report MSU-940722, Mississippi State University — Dept. of Computer Science, August 1994. Archived at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.6283. 6.2.2	41 42 43 44
[64]	Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. <i>Parallel Computing</i> , 20(4):565–596, April 1994. 7.1.2, 7.5.6	45 46 47 48

[65] The Internet Society. XDR: External Data Representation Standard, May 2006. http: $\mathbf{2}$ //www.rfc-editor.org/pdfrfc/rfc4506.txt.pdf. 14.5.2 [66] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. Scientific Programming, 5(4):301–317, Winter 1996. 14.1 [67] Jesper Larsson Träff. SMP-aware message passing programming. In Eighth Inter-national Workshop on High-level Parallel Programming Models and Supportive Envi-ronments (HIPS), 17th International Parallel and Distributed Processing Symposium (*IPDPS*), pages 56–65, 2003. 7.4.2 [68] The Unicode Standard, Version 2.0. Addison-Wesley, 1996. ISBN 0-201-48345-9. 14.5.2 [69] D. Walker. Standards for message passing in a distributed memory environment. Tech-nical Report TM-12147, Oak Ridge National Laboratory, August 1992. 1.2 24 31

General Index

This index lists mainly terms of the MPI specification. The underlined page numbers refer to the definitions or parts of the definition of the terms. Bold face numbers mark section titles.

absolute addresses, 20, <u>128</u>, 767 access epoch, 542action in function names, $\underline{10}$ active, 62, 350 active target communication, 542 addresses, 143 absolute, 20, <u>128</u>, 767 correct use, 143 relative displacement, 20, 128 alignment, 412, 511, 513, 943 all-reduce, 218 nonblocking, 242 persistent, 261 all-to-all, 198 nonblocking, 238 persistent, 256 array arguments, 18 assertions, 556 ASYNCHRONOUS Fortran attribute, 768 attribute, 277, 328, 787 caching, $\underline{276}$ backward incompatibilies, 729 barrier synchronization, 177 nonblocking, 230 persistent, 248 blocking, <u>44</u>, 47, 607 I/O, 608 Blocking operation, 12 Blocking procedure, $\underline{14}$ bounds of datatypes, 134 broadcast, 178 nonblocking, 230 persistent, 249 buffer allocation, 52 buffered, <u>44</u>, <u>55</u>, <u>56</u> nonblocking, 55 buffered send, 48

language binding, 23	12
caching, 275 , <u>276</u> , 328	13
callback functions	14
language interoperability, 786	15
prototype definitions, 810	16
deprecated, 816	17
$(a_1, 22, 03, 10, 04, 452, 124, 542)$	18
canonical pack and unpack, 166	
Cartesian	19
topology, <u>354</u> , 356	20
Chameleon, 2	21
change-log, 941	22
Chimp, 2 choice, 19	23
class	24
in function names, $\underline{10}$	25
clock synchronization, 410	26
collective, 607	27
split, 648	28
collective communication, 171	20
correctness, 265	
file data access operations, 626	30
neighborhood, 379	31
nonblocking, 228	32
Collective operation, $\underline{12}$	33
Collective procedure, 14	34
commit, $\underline{137}$	35
COMMON blocks, 771	36
communication, 505	37
collective, 171	38
modes, 44	39
one-sided, 505	40
overlap with computation, 55	
partitioned point-to-point, 93	41
point-to-point, 27	42
RMA, <u>505</u>	43
communicator, $\underline{32}$, 275 , $\underline{276}$	44
hidden, 174 , 268 , 550	45
complete operation, $\underline{12}$	46
Completing procedure, $\underline{14}$	47
completion, 62	48

General Index

968

1 multiple, 67 $\mathbf{2}$ completion stage, $\underline{12}$ connected, 5003 constants, 18, 791, 795 4 context, 275, 276, 278 5control variables 6 tools interface, 678 7 conversion, 428 counts, 20 9 create 10 in function names, 10 11 data, 29 12data conversion, 42 13 datatype 14derived, 15 15equivalent, 1516named, 15 portable, 1517predefined, 15 18 unnamed, $\underline{15}$ 19datatypes, 109, 785 20deadlock 21avoid, nonblocking communication, 54 22 default file error, 659 23delete in function names, 10 24deprecated interfaces, 21, 721 25derived datatype, 15, 109, 763 26disconnected, 501 27displacement, 591, 605 28distributed graph 29 topology, 354, 360 30 Dynamic Process Model, 443 31 dynamically attached memory, **515** 32 elementary datatype, 591, 606 33 empty, 62 34 end of file, 59335 envelope, 27, 31 36 environmental inquiries, 409 37 equivalent datatypes, 15error handling, 24, 414 38 default file error handler, 594, 597, 659 39 error codes and classes, 425, 426 40 error handlers, 417, 426, 787 41 finalize, 25, 452, 453, 501 42I/O, 658, 659 43 initial error handler, 25, 415, 445, 453, 485 44 one-sided communication, 558 process failure, 24, 501 45program error, 24 46 resource error, 24 47startup, 25, 445, 485 48

transmission failure, $\underline{24}$ establishing communication, 487etype, $\underline{591}$, 606 events tools interface, 697exclusive scan nonblocking, 246persistent, 265explicit offsets, $\underline{608}$, 610exposure epoch, $\underline{542}$ extent of datatypes, $\underline{110}$, $\underline{133}$, $\mathbf{134}$ true extent, $\mathbf{136}$ external32file data representation, $\underline{639}$ extra-state, 791

fairness, 49, <u>73</u> file, <u>591</u> data access, **607** collective operation

collective operations, 626 explicit offsets, 610 individual file pointers, 615 seek, **628** shared file pointers, 623 split collective, 630 end of file, 593 filetype, 592handle, 593 interoperability, 637 manipulation, 593 offset, 20, 592 pointer, 593 size, 593 view, 591, 592, 604 file size, 654 finalize, 450 finished, 453 Fortran language binding, 21, 731 Fortran support, 731 Freeing procedure, 14 freeing stage, 12

gather, **179** nonblocking, **232** persistent, **250** gather-to-all, **195** nonblocking, **236** persistent, **254** general datatype, <u>109</u> generalized requests, **581**, <u>581</u> get in function names, <u>10</u> graph

topology, <u>354</u>, **358** group, 275, 276, 278, 319 group objects, 278handles, <u>16</u>, **780** hardware resource type, 303, 304 host rank, 409 immediate, <u>14</u>, <u>56</u> inactive, 62 inclusive scan, 224 nonblocking, 245 persistent, 264 incomplete, <u>14</u>, <u>56</u> Incomplete procedure, 14 independent, 501individual file pointers, 608, 615 info object, 435 file info, 601keys, 817 values, 818 initial error handler, 25Initialization procedure, $\underline{13}$ initialization stage, $\underline{11}$ initiation, 11, 56 Initiation procedure, 14inter-communication, 277, 318 inter-communicator, <u>277</u>, <u>318</u> collective operations, 175, 176 interlanguage communication, 792 internal file data representation, 638 interoperability, 637 intra-communication, 277, 318 intra-communicator, 276, 318 collective operations, 174 intra-communicator objects, $\underline{279}$ I/O, **591** IO rank, **410** isin function names, 10language binding, 21, 731 interoperability, 779 summary, 795 lb_marker, 122, 126, 132, 137 erased, 135 local, 44local group, 292Local procedure, $\underline{13}$ loosely synchronous model, 350 lower bound, 132lower-bound markers, 132

macros, 23

	main thread, 473	1
	matched receives, 81	2
	matching	3
	type, 40 , <u>140</u> , 653	4
	matching probe, 79	5
	memory	
	alignment, 412, 511, 513, 943	6
	allocation, 411 , 509, 511	7
	system, $\underline{16}$	8
	memory model, 506 , 541	9
	separate, 506, 513	10
	unified, 506, 513	11
	message, 27	12
	data, 29	13
	envelope, 31	14
	modes, 44	
	module variables, 771	15
	mpi module	16
	Fortran support, 735	17
	MPI operation, $\underline{11}$	18
	MPI process initialization, $\underline{443}$	19
	Dynamic Process Model, 443	20
	Sessions Model, 279 , 282 , 409 , 443 , 473 ,	21
	594	22
	World Model, 279, 282, 409, 443, 473,	23
	594, 659	20
	mpi_f08 module	
	Fortran support, 732	25
	MPI_SIZEOF and storage_size(), 22, 725,	26
Ì	755–757	27
1	mpiexec, 446 , 469 , 470	28
	mpif.h include file	29
	Fortran support, 737	30
	mpirun, 470 multiple completions 67	31
	multiple completions, 67	32
	named datatype, $\underline{15}$	
	names, 488	33
	name publishing, 493	34
	naming objects, 345	35
	native	36
	file data representation, $\underline{638}$	37
	neighborhood collective communication, 379	38
	nonblocking, 391	39
	periodic and dims==1 or 2, 942	40
	non-local, 44, 45	41
	Non-local procedure, $\underline{13}$	42
	nonblocking, 54 , 522, 607	
	communication, $\underline{54}$	43
	completion, 62	44
	Fortran problems, 766	45
	I/O, 608	46
	initiation, 56	47
	persistent	48

General Index

1 partitioned completion, 100 $\mathbf{2}$ request objects, 56 Nonblocking operation, 123 Nonblocking procedure, 144 Noncollective operation, 12 5null handle, 62 6 null processes, 91 7 8 offset, 20, 592 one-sided communication, 505 9 Fortran problems, 767 10 opaque objects, 16, 785 11 operation 12blocking, 12 13 collective, 12 14complete, 1215nonblocking, 1216noncollective, 12operation-related, 1317partitioned receive, 12 18 partitioned send, $\underline{12}$ 19 persistent, 1220origin, 506 21pack, 159 22 canonical, 166 23packing unit, 162 24parallel procedure, 35025partitioned completion, 100 26partitioned point-to-point communication, 93 27passive target communication, 54228performance variables. 29 tools interface, **685** persistent communication requests, 85 30 collective persistent, 246, 396 31 Fortran problems, 767 32 Persistent operation, 12 33 PICL, 2 34PMPI_. 665 35 point-to-point communication, 27 36 portable datatype, 1537 ports, **488** POSIX 38 environment, 433 39 FORTRAN, 17 40 I/O, 594, 595, 608, 638, 649 41 model, **591** 42predefined datatype, 15 43 predefined reduction operations, 206 44private window copy, 540 probe, 76 45probe, matching, 79 46 procedure 47blocking, 14 48

collective, 14 completing, 14freeing, 14incomplete, 14 initialization, 13 initiation, 14 local, 13 non-local, $\underline{13}$ nonblocking, 14 operation-related, 13 starting, 13 process creation, 443 process failures, 24 process group, $\underline{32}$ processes, 23 processor name, 410 profiling interface, 665 program error, 24 progress, 229 prototype definitions, 810 deprecated, 816 public window copy, 540 PVM, 2

rank, 278 ready, <u>45</u>, <u>55</u>, <u>56</u> nonblocking, 55ready send, 47 receive, 27, 28, 32 buffer, 28 complete, 55 context, 319 start call, 55reduce, **204** nonblocking, 241 persistent, 260 reduce-scatter, 221 nonblocking, 243, 244 persistent, 262, 263 reduction operations, 204, 787 predefined. 206 process-local, 219 scan, 224 user-defined, 213 related, 162 relative displacement, 20, 128 remote group, 292 Remote Memory Access see RMA, 505 removed interfaces, 21, 727 request complete I/O, 608 request objects, 56 resource error, 24

RMA, 505 communication calls, **522** request-based, 535 memory model, 540 synchronization calls, 542 scan. 224 inclusive, 224 scatter, 189 nonblocking, 234 persistent, 252 seek, 628 semantics, 941 collective communications, 172 file collective, 652 collective access, 626 conflicting access, 649 consistency, 648 explicit offsets, 610 nonblocking collective, 652 shared file pointer, 624 split collective, 648 inter-communicator. 292 MPI_COMM_IDUP, 294 nonblocking communications, 66 nonblocking partitioned communications, 101 partitioned point-to-point communication, 94 point-to-point communication, 48 process failure, **501** semantics and correctness one-sided communication, 558 send, <u>27</u>, 28 buffer, 27complete, 55 context, 319 start, 55 send-receive, 37 nonblocking, 61, 62 separate memory model, 506, 513, 541 sequential storage, 143serialization, 55 Sessions Model, 279, 282, 409, 443, 473, 594 set in function names, 10 shared file pointers, 608, 623 shared memory allocation, 511 signals, 26 singleton init, 499 size changing I/O, 654 source, 319

1 split collective, 607, 630 2 stage completion, $\underline{12}$ freeing, 12 initialization, 11 5 starting, 11 6 standard, 44, 55 nonblocking, 55standard send, 48 started, 453 Starting procedure, 1310 starting processes, 476, 477 11 starting stage, 11 12startup, 444 13 portable, 470 14state, 18 15status, **34**, **782** 16associating information, 588 ignore, 37 17 test, **75** 18 strong synchronization, 543 19synchronization, 505, 522 20synchronization calls 21RMA, **542** 22 synchronous, <u>45</u>, 55, <u>56</u>, <u>62</u> 23nonblocking, 55synchronous send, 47 24 system memory, 16 2526tag values, 409 27target, 506 28thread compliant, 448, 472 29threads, 472 30 thread-safe, 2, 408, 425, 426, 701 timers and synchronization, 433 31 tool information interface, 671 32 tool support, 665 33 topologies, 353 34 topology 35 Cartesian, <u>354</u>, **356** 36 distributed graph, 354, 360 37 graph, 354, 358 virtual. 354 38 transmission failures, 24 39 true extent of datatypes, 136 40 type map, 11041 type matching, 40, 14042type signature, 110 43 types, 809 44 ub_marker, 122, 126, 127, 132, 137 45erased, 135 46 unified memory model, 506, 513, 541 47universe size, 498 48

```
1
       unnamed datatype, \underline{15}
\mathbf{2}
       \mathrm{unpack},\, \mathbf{159}
            canonical, 166
3
       upper bound, \underline{132}
4
       upper-bound markers, 132
\mathbf{5}
       user functions at process termination, 454
6
       user-defined data representations, 644
7
       user-defined reduction operations, {\bf 213}
8
       verbosity levels
9
            tools interface, 672
10
       version inquiries, 407
11
       view, 591, <u>592</u>, 604
12
       virtual topology, 276, 277, 354
13
14
       weak synchronization, 543
       window
15
            allocation, 509
16
            creation, 507
17
            dynamically attached memory, 515
18
            shared memory allocation, 511
19
       World Model, 279, 282, 409, 443, 473, 594,
20
                  659
21
       XDR, 43
22
23
       {\rm Zipcode},\, {\color{black} 2}
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

Examples Index

This index lists code examples throughout the text. Some examples are referred to by content; others are listed by the major MPI function that they are demonstrating. MPI functions listed in all capital letter are Fortran examples; MPI functions listed in mixed case are C examples.

ASYNCHRONOUS, 575, 775, 777	register optimization, 766–768
Attributes between languages, 788	
	Independence of nonblocking operations, 272
Blocking/Nonblocking collectives do not	Intercommunicator, 296, 300
match, 270	Interlanguage communication, 792
	Intertwined matching pairs, 49
C/Fortran handle conversion, 781	
Cartesian virtual topologies, 402	Message exchange (ping-pong), 50
Client-server code, 74	Mixing blocking and nonblocking collective
with probe, 78	operations, 268
with probe, wrong, 78	Mixing collective and point-to-point requests,
	271
Datatype	MPI_ACCUMULATE, 530
3D array, 151	MPI_Accumulate, 573, 574, 577
absolute addresses, 156	MPI_Aint, 153
array of structures, 153	MPI_Aint_add, 577
elaborate example, 164, 165	MPI_Allgather, 198
matching type, 140	MPI_ALLOC_MEM, 413, 414
matrix transpose, 152	MPI_Alloc_mem, 414, 577
union, 157	MPI_ALLREDUCE, 219
Datatypes	MPI_Alltoall, 270
matching, 41	MPI_ASYNC_PROTECTS_NONBLOCKING,
not matching, 41	575
untyped, 41	MPI_BARRIER, 466
Deadlock	MPI_Barrier, 452, 563–566, 572–574
if not buffered, 51	MPI_Bcast, 179, 266–269
with MPI_Bcast, 266	MPI_BSEND, 48, 49
wrong message exchange, 50	MPI_Buffer_attach, 53, 452
	MPI_Buffer_detach, 53
False matching of collective operations, 269	MPI_BYTE, 41
Fortran 90	MPI_Cancel, 452
copying and sequence problem, 759, 761,	MPI_CART_COORDS, 376
762	MPI_CART_GET, 402
derived types, 763	MPI_CART_RANK, 376
heterogeneous communication (unsafe),	MPI_CART_SHIFT, 376, 386, 402
756	MPI_CART_SUB, 377
invalid KIND, 752	MPI_CARTDIM_GET, 386
MPI_TYPE_MATCH_SIZE	MPI_CHARACTER, 42
implementation, 755	MPI_Comm_accept, 496, 497
overlapping communication and	MPI_Comm_connect, 496, 497
computation, 775, 776, 778	MPI_Comm_create, 296, 312, 313, 316

1	MPI_COMM_CREATE_FROM_GROUP, 462,
2	466
3	MPI_Comm_create_keyval, 343
4	MPI_Comm_dup, 315
5	MPI_Comm_get_attr, 343
6	$MPI_Comm_get_parent, 486$
	MPI_Comm_group, 296, 316, 343
7	MPI_Comm_remote_size, 300
8	MPI_Comm_set_attr, 343
9	MPI_COMM_SPAWN, 479
10	MPI_Comm_spawn, 479, 486
11	MPI_COMM_SPAWN_MULTIPLE, 484
12	MPI_Comm_spawn_multiple, 484
13	MPI_Comm_split, 300, 326, 327
14	MPI_Comm_split_type, 304, 305
15	MPI_Compare_and_swap, 574, 577
	MPI_DIMS_CREATE, 357, 402
16	MPI_DIST_GRAPH_CREATE, 365
17	MPI_Dist_graph_create, 366
18	MPI_DIST_GRAPH_CREATE_ADJACENT,
19	365 MDLE
20	MPI_F_sync_reg, 575
21	MPI_FILE_CLOSE, 616, 619 MPI_FILE_GET_AMODE, 600
22	
23	MPI_FILE_IREAD, 619 MPI_FILE_OPEN, 616, 619
24	MPI_FILE_READ, 616
25	MPI_FILE_SET_ATOMICITY, 654
	MPI_FILE_SET_VIEW, 616, 619
26	MPI_FILE_SYNC, 655
27	MPI_Finalize, 451–453
28	MPI_FREE_MEM, 413, 414
29	MPI_Free_mem, 577
30	MPI_Gather, 165, 182, 183, 187
31	MPI_Gatherv, 165, 184–187
32	MPI_GET, 526, 528
33	MPI_Get, 563–566, 571, 572
34	MPI_Get_accumulate, 573, 574, 577
35	MPI_GET_ADDRESS, 129, 763, 764, 785
	MPI_Get_address, 153, 156, 157, 164
36	MPI_GET_COUNT, 142
37	MPI_GET_ELEMENTS, 142
38	MPI_GRAPH_CREATE, 359, 372
39	MPI_GRAPH_NEIGHBORS, 372
40	MPI_GRAPH_NEIGHBORS_COUNT, 372
41	MPI_Grequest_complete, 586
42	MPI_Grequest_start, 586
43	MPI_Group_excl, 312
44	MPI_Group_free, 296, 312, 313
45	MPI_GROUP_FROM_SESSION_PSET, 462,
	464, 466
46	MPI_Group_incl, 296, 313, 316
47	MPI_Iallreduce, 271
48	MPI_Ialltoall, 270

MPI_Ibarrier, 268–271 MPI_Ibcast, 231, 271, 272 MPI_INFO_CREATE, 462 MPI_INFO_ENV, 446 MPI_INFO_SET, 462 MPI_Intercomm_create, 326, 327 MPI_Iprobe, 452 MPI_IRECV, 64, 66, 67, 74 MPI_Irecv, 271 MPI_ISEND, 64, 66, 74 MPI_NEIGHBOR_ALLGATHER, 382 MPI_NEIGHBOR_ALLTOALL, 386 MPI_Op_create, 216, 217, 226 MPI_Open_port, 496, 497 MPI_Pack, 163–165 MPI_Pack_size, 165 MPI_PROBE, 78 MPI_Publish_name, 496 MPI_Put, 548, 554, 564-566, 571, 572 MPI_RECV, 41, 42, 48–51, 67, 78, 140 MPI_Recv, 270 MPI_REDUCE, 208, 211 MPI_Reduce, 211, 212, 216, 217 MPI_REQUEST_FREE, 66 MPI_Request_free, 451 MPI_Rget, 576 MPI_Rput, 576 MPI_Scan, 226 MPI_Scatter, 192 MPI_Scatterv, 193, 194 MPI_SEND, 41, 42, 50, 51, 67, 78, 140 MPI_Send, 153, 156, 157, 164, 270, 271 MPI_SENDRECV, 151, 152, 386 MPI_SENDRECV_REPLACE, 376 MPI_SESSION_FINALIZE, 462, 464, 466 MPI_SESSION_GET_NTH_PSET, 464, 466 MPI_SESSION_GET_NUM_PSETS, 464, 466 MPI_SESSION_INIT, 462, 464, 466 MPI_SSEND, 49, 67 MPI_Test_cancelled, 452 MPI_TYPE_COMMIT, 138, 151, 152, 526, 763, 764 MPI_Type_commit, 153, 156, 157, 164, 183-187, 194, 226 MPI_TYPE_CONTIGUOUS, 112, 132, 140, 142MPI_Type_contiguous, 183 MPI_TYPE_CREATE_DARRAY, 128 MPI_TYPE_CREATE_HVECTOR, 151, 152 MPI_Type_create_hvector, 153, 156 MPI_TYPE_CREATE_INDEXED_BLOCK, 526MPI_TYPE_CREATE_RESIZED, 763, 764

MPI_TYPE_CREATE_STRUCT, 120, 132. 152, 763, 764 MPI_Type_create_struct, 153, 186, 187, 226 MPI_TYPE_CREATE_SUBA MPI_TYPE_EXTENT, 526 MPI_TYPE_FREE, 526 MPI_Type_get_contents, 158 MPI_Type_get_envelope, 158 MPI_TYPE_GET_EXTENT, 530MPI_Type_get_extent, 153 MPI_TYPE_INDEXED, 116, MPI_Type_indexed, 153, 156 MPI_TYPE_VECTOR, 113, 1 MPI_Type_vector, 184, 185, 18 MPI_Unpack, 164, 165 MPI_Unpublish_name, 496 MPI_User_function, 217 MPI_WAIT, 64, 66, 67, 74, 619 MPI_Wait, 268-271 MPI_Waitall, 271, 576 MPI_WAITANY, 74 MPI_Waitany, 576 MPI_WAITSOME, 74 MPI_Win_attach, 577 MPI_Win_complete, 548, 566, MPI_WIN_CREATE, 526, 528 MPI_Win_create_dynamic, 57 MPI_Win_detach, 577 MPI_WIN_FENCE, 526, 528, MPI_Win_fence, 571 MPI_Win_flush, 565, 573, 574, MPI_Win_flush_all, 574 MPI_Win_flush_local, 564 MPI_WIN_FREE, 528, 530 MPI_Win_lock, 554, 563-566 MPI_Win_lock_all, 575-577 MPI_Win_post, 566, 572 MPI_Win_start, 548, 566, 572 MPI_Win_sync, 564, 565, 573, shared memory windows, MPI_Win_unlock, 554, 563-56 MPI_Win_unlock_all, 576, 577 MPI_Win_wait, 566, 572 mpiexec, 446, 471, 472 Neighborhood collective comm Non-deterministic program wit 267Non-overtaking messages, 48 Nonblocking operations, 64, 66

message ordering, 66

progress, 67

156, 157, 164,Pipelining nonblocking collective operation271RRAY, 663Profiling interfaceimplementation union the Communication	
RRAY, 663 Profiling interface	ns,
 implementation using the C macro preprocessor, 669 implementation using weak symbols, measurement wrapper, 668 151, 152, 528, Progression of nonblocking collective operations, 270 	668
151 Shared memory windows MPI_Win_sync, 575	
51, 152 57, 194 Threads and MPI, 473 Tool information interface basic usage of performance variables, listing names of all control variables, reading the value of a control variable Topologies, 402 Typemap, 111–113, 116, 120, 128	681
Virtual topologies, 402	
572 530 577	
574	
575 6	

1 2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

 31

32

33

34

35

36

37

38

39

40

41 42

43

44

45

46

47

MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles, including info keys and values. Underlined page numbers give the location of the primary definition or use of the indexed term.

```
14
      "access_style", 603
15
      "accumulate_ops", 508
16
      "accumulate_ordering", 508, 567, 568
      "alloc_shared_noncontig", 512, 513
17
      "appnum", 500
18
      "arch", 446, 485
19
      "argv", 446
20
21
      "cb_block_size", 603
22
      "cb_buffer_size", 603
      "cb_nodes", 603
23
      "chunked", 603
24
      "chunked_item", 603
25
      "chunked_size", 603
26
      "collective_buffering", 603
27
      "command", 446
28
      external32, 639
29
30
      "false", 309, 603
^{31}
      "file", 446, 485
32
      "file_perm", 603
33
      "filename", 603
34
      "host", 446, 485
35
      "hwloc://L3Cache", 458
36
37
      internal, 638
38
      "io_node_list", 604
39
      "ip_address", 496
      "ip_port", 495
40
41
      "maxprocs", 446, 447
42
      "mpi://", 458
43
      "mpi://SELF", 458, 459, 594
44
      "mpi://WORLD", 458, 459, 464
45
      MPI::_LONG_LONG, 947
      MPI::BOOL, 947
46
      MPI::COMPLEX, 947
47
      MPI::DOUBLE_COMPLEX, 947
48
```

5 6

12

13

MPI::F_COMPLEX16, 947 MPI::F_COMPLEX32, 947 MPI::F_COMPLEX4, 947 MPI::F_COMPLEX8, 947 MPI::INTEGER16, 947 MPI::LONG_DOUBLE_COMPLEX, 947 MPI::LONG_LONG, 947 MPI::REAL16, 947 MPI_2DOUBLE_PRECISION, 210, 801 MPI_2INT, 210, 801 MPI_2INTEGER, 210, 801 MPI_2REAL, 210, 801 MPI_ADDRESS_KIND, 19, 20, 31, 329, 758, 788, 798 MPI_AINT, 31, 207, 516, 642, 799, 800, 952-955 MPI_ANY_SOURCE, 33, <u>34</u>, 38, 39, 48, 59–62, 76-81, 89, 309, 351, 410, 797 MPI_ANY_TAG, 18, 33, 34, 36, 38, 39, 59-62, 76, 77, 79-83, 89, 91, 309, 797, 949 MPI_APPNUM, 500, 804 MPI_ARGV_NULL, 19, 479, 480, 484, 758, 806 MPI_ARGVS_NULL, 19, 483, 758, 806 "mpi_assert_allow_overtaking", 309, 943 "mpi_assert_exact_length", 309, 943 "mpi_assert_no_any_source", 309, 943 "mpi_assert_no_any_tag", 309, 943 MPI_ASYNC_PROTECTS_NONBLOCKING, 19, 575, 732, 733, 735, 737, 740, 747, 749, 769, 798, 952 MPI_BAND, 206, 208, 802 MPI_BOR, 206, 208, 802 MPI_BOTTOM, 10, 19, 20, 37, 101, 128, 143, 144, 174, 362, 364, 481, 515, 520, 734, 736, 743, 758, 763, 765, 767, 768, 770-772, 774, 785, 786, 792, 797, 959 MPI_BSEND_OVERHEAD, 54, 797 MPI_BXOR, 206, 208, 802 MPI_BYTE, 29, 30, 40, 41, 43, 166, 207, 592,

638, 639, 642, 653, 792, 799, 800, 955 MPI_C_BOOL, 30, 207, 642, 799, 947, 952-955 MPI_C_COMPLEX, 30, 207, 642, 799, 947, 953-955 MPI_C_DOUBLE_COMPLEX, 30, 207, 642, 799, 953–955 MPI_C_FLOAT_COMPLEX, 30, 207, 642, 799, 953-955 MPI_C_LONG_DOUBLE_COMPLEX, 30, 207, 642, 799, 953–955 MPI_CART, 367, 803 MPI_CHAR, <u>30</u>, 43, 120, 209, 642, 676, 799, 952 MPI_CHARACTER, 29, 42, 43, 209, 642, 800 MPI_COMBINER_CONTIGUOUS, 145, 148, 805 MPI_COMBINER_DARRAY, 145, 150, 805 MPI_COMBINER_DUP, 145, 148, 805 MPI_COMBINER_F90_COMPLEX, 145, 150, 805 MPI_COMBINER_F90_INTEGER, 145, 150, 805 MPI_COMBINER_F90_REAL, 145, 150, 805 MPI_COMBINER_HINDEXED, 22, 145, 149, 805 MPI_COMBINER_HINDEXED_BLOCK, 145, 149, 805, 949 MPI_COMBINER_HINDEXED_INTEGER, 22, 728, 948 MPI_COMBINER_HVECTOR, 22, 145, 149, 805 MPI_COMBINER_HVECTOR_INTEGER, 22, 728, 948 MPI_COMBINER_INDEXED, 145, 149, 805 MPI_COMBINER_INDEXED_BLOCK, 145, 149, 805 MPI_COMBINER_NAMED, 145, 148, 805 MPI_COMBINER_RESIZED, 145, 151, 805 MPI_COMBINER_STRUCT, 22, 145, 150, 805 MPI_COMBINER_STRUCT_INTEGER, 22, 728, 948 MPI_COMBINER_SUBARRAY, 145, 150, 805 MPI_COMBINER_VECTOR, 145, 148, 805 MPI_COMM_DUP_FN, 22, 332, 803, 951 MPI_COMM_NULL, 279, 295, 296, 298-300, 303, 304, 308, 325, 347, 356, 358, 481, 482, 502, 503, 780, 802, 956 MPI_COMM_NULL_COPY_FN, 22, 332, 734, 787, 803, 951 MPI_COMM_NULL_DELETE_FN, 22, <u>332</u>, 803 MPI_COMM_PARENT, 347 MPI_COMM_SELF, 25, 279, 298, 310, 329, 347, 415, 444, 445, 450, 452, 454, 455, 502, 594, 729, 801, 943, 954 MPI_COMM_TYPE_HW_GUIDED, 303, 304, 801, 942 MPI_COMM_TYPE_HW_UNGUIDED, <u>304</u>, 305, 307, 801, 942

MPI_COMM_TYPE_SHARED, <u>303</u> , 304, 801,	1
949	2
MPI_COMM_WORLD, 18, 25, 32, 277,	3
279-282, 290, 291, 304, 305, 307, 310,	4
313, 322, 347, 357, 409, 410, 418, 429,	5
443-446, 448-451, 453, 455, 469, 470,	6
476-478, 482, 484, 498-502, 637, 658,	7
659,682,691,729,780,791,801,943,	
957	8
MPI_COMPLEX, <u>29</u> , 207, 641, 642, 750, 800	9
MPI_COMPLEX16, <u>30</u> , 207, 643, 800	10
MPI_COMPLEX32, <u>30</u> , 207, 643, 800	11
MPI_COMPLEX4, <u>29</u> , 207, 643, 800	12
MPI_COMPLEX8, 29, 207, 643, 800	13
MPI_CONGRUENT, <u>291</u> , 321, 801	14
MPI_CONVERSION_FN_NULL, 647, 803	
MPI_COUNT, <u>31</u> , 207, 642, 676, 799, 800, 948	15
MPI_COUNT_KIND, 19, <u>20</u> , 31, 798	16
MPI_CXX_BOOL, <u>31</u> , 207, 642, 643, 800, 947	17
MPI_CXX_DOUBLE_COMPLEX, <u>31</u> , 207, 642,	18
643,800,947	19
MPI_CXX_FLOAT_COMPLEX, <u>31</u> , 207, 642,	20
643,800,947	21
MPI_CXX_LONG_DOUBLE_COMPLEX, <u>31</u> ,	22
207,642,643,800,947	
MPI_DATATYPE_NULL, 138, 802	23
MPI_DISPLACEMENT_CURRENT, 605, 806,	24
958	25
MPI_DIST_GRAPH, 367, 803, 954	26
MPI_DISTRIBUTE_BLOCK, 125, 806	27
MPI_DISTRIBUTE_CYCLIC, 125, 806	28
MPI_DISTRIBUTE_DFLT_DARG, 125, 806	29
MPI_DISTRIBUTE_NONE, 125, 806	30
MPI_DOUBLE, <u>30</u> , 207, 642, 676, 686, 687,	
749, 799	31
MPI_DOUBLE_COMPLEX, <u>29</u> , 207, 641, 642,	32
750, 800	33
MPI_DOUBLE_INT, 210, 211, 801	34
MPI_DOUBLE_PRECISION, <u>29</u> , 207, 642, 750,	35
800 MDI DUD EN 22 222 722 204	36
MPI_DUP_FN, 22, 332, <u>722</u> , 804	37
MPI_ERR_ACCESS, 427, 597, 660, 796	38
MPI_ERR_AMODE, 427, 595, 660, 796 MPI_ERR_ARG, 427, 795	39
MPI_ERR_ASSERT, 427, 559, 796	
MPI_ERR_BAD_FILE, 427, 660, 796	40
MPI_ERR_BASE, 413, 427, 559, 796	41
MPI_ERR_BUFFER, 427, 795	42
MPI_ERR_COMM, 427, 795	43
MPI_ERR_CONVERSION, 427, 647, 660, 796	44
MPI_ERR_COUNT, 427, 795	45
MPI_ERR_DIMS, 427, 795	46
MPI_ERR_DISP, 427, 559, 796	47
MPI_ERR_DUP_DATAREP, 427, 645, 660, 796	48
, .,,,,	

1 MPI_ERR_FILE, 427, 660, 796 $\mathbf{2}$ MPI_ERR_FILE_EXISTS, 427, 660, 796 MPI_ERR_FILE_IN_USE, 427, 597, 660, 796 3 MPI_ERR_GROUP, 427, 795 4 MPI_ERR_IN_STATUS, 35, 37, 63, 70, 72, 417, 5425, 427, 585, 610, 796 6 MPI_ERR_INFO, 427, 796 $\overline{7}$ MPI_ERR_INFO_KEY, 427, 437, 796 8 MPI_ERR_INFO_NOKEY, 427, 437, 796 9 MPI_ERR_INFO_VALUE, 427, 437, 796 10 MPI_ERR_INTERN, 416, 427, 795 MPI_ERR_IO, 427, 660, 796 11 MPI_ERR_KEYVAL, 343, 427, 796 12MPI_ERR_LASTCODE, 426, 428-430, 718, 797 13MPI_ERR_LOCKTYPE, 427, 559, 796 14MPI_ERR_NAME, 427, 495, 796 15MPI_ERR_NO_MEM, 412, 427, 796 16MPI_ERR_NO_SPACE, 427, 660, 796 17MPI_ERR_NO_SUCH_FILE, 427, 597, 660, 796 MPI_ERR_NOT_SAME, 427, 660, 796 18 MPI_ERR_OP, 428, 558, 795 19MPI_ERR_OTHER, 426, 428, 795 20MPI_ERR_PENDING, 70, 428, 795 21MPI_ERR_PORT, 428, 492, 796 22MPI_ERR_PROC_ABORTED, 428, 469, 796, 23943 24 MPI_ERR_QUOTA, 428, 660, 796 25MPI_ERR_RANK, 428, 558, 795 MPI_ERR_READ_ONLY, 428, 660, 796 26MPI_ERR_REQUEST, 428, 795 27MPI_ERR_RMA_ATTACH, 428, 559, 796 28MPI_ERR_RMA_CONFLICT, 428, 559, 796 29MPI_ERR_RMA_FLAVOR, 428, 514, 559, 796 30MPI_ERR_RMA_RANGE, 428, 559, 796 31 MPI_ERR_RMA_SHARED, 428, 559, 796 32 MPI_ERR_RMA_SYNC, 428, 559, 796 MPI_ERR_ROOT, 428, 795 33 MPI_ERR_SERVICE, 428, 495, 796 34MPI_ERR_SESSION, 428 35 MPI_ERR_SIZE, 428, 559, 796 36 MPI_ERR_SPAWN, 428, 480, 481, 796 37 MPI_ERR_TAG, 428, 795 38MPI_ERR_TOPOLOGY, 428, 795 39 MPI_ERR_TRUNCATE, 428, 795 MPI_ERR_TYPE, 428, 659, 795 40MPI_ERR_UNKNOWN, 426, 428, 795 41 MPI_ERR_UNSUPPORTED_DATAREP, 428, 42660, 796 43MPI_ERR_UNSUPPORTED_OPERATION, 428, 44660, 796 45MPI_ERR_WIN, 428, 559, 796 46MPI_ERRCODES_IGNORE, 19, 481, 758, 806 47MPI_ERRHANDLER_NULL, 424, 802 48

MPI_ERROR, 34, <u>35</u>, 63, 228, 535, 782, 798, 944, 951 MPI_ERRORS_ABORT, 415, 445, 485, 943 "mpi_errors_abort", 485, 944 MPI_ERRORS_ARE_FATAL, 415, 416, 432, 485, 558, 658, 798, 943 "mpi_errors_are_fatal", 485, 944 MPI_ERRORS_RETURN, 415, 416, 432, 469, 485, 659, 791, 798 "mpi_errors_return", 485, 944 MPI_F08_STATUS_IGNORE, 784, 807, 951 MPI_F08_STATUSES_IGNORE, 784, 807, 951 MPI_F_ERROR, <u>782</u>, 798, 941 MPI_F_SOURCE, 782, 798, 941 MPI_F_STATUS_IGNORE, 782, 807 MPI_F_STATUS_SIZE, 19, 782, 798, 941 MPI_F_STATUSES_IGNORE, 782, 807 MPI_F_TAG, 782, 798, 941 MPI_FILE_NULL, 596, 659, 802 MPI_FLOAT, 30, 120, 205, 207, 640, 642, 799 MPI_FLOAT_INT, 15, 210, 211, 801 MPI_GRAPH, 367, 803 MPI_GROUP_EMPTY, 278, 284, 285, 295, 296, 298, 306, 325, 802 MPI_GROUP_NULL, 278, 288, 289, 802 MPI_HOST, 409, 801 "mpi_hw_resource_type", 303-305, 942 MPI_IDENT, 282, 291, 801 MPI_IN_PLACE, 19, 174, 201, 737, 758, 797 MPI_INFO, 248 MPI_INFO_ENV, 442, 446, 447, 455, 801, 950 MPI_INFO_NULL, 365, 412, 441, 480, 490, 595, 597, 606, 802 "mpi_initial_errhandler", 445, 446, 485, 944 MPI_INT, 15, <u>30</u>, 110, 207, 640-642, 675, 676, 679, 685, 689, 749, 791, 793, 799 MPI_INT16_T, 30, 207, 642, 799, 952-955 MPI_INT32_T, 30, 207, 642, 676, 799, 952-955 MPI_INT64_T, 30, 207, 642, 676, 799, 952-955 MPI_INT8_T, 30, 207, 642, 799, 952-955 MPI_INTEGER, 29, 40, 207, 642, 749, 750, 793, 800 MPI_INTEGER1, 29, 207, 643, 800 MPI_INTEGER16, 207, 643, 800 MPI_INTEGER2, 29, 207, 641, 643, 800 MPI_INTEGER4, 29, 207, 643, 800 MPI_INTEGER8, 29, 207, 643, 754, 800 MPI_INTEGER_KIND, 19, 798 MPI_IO, 409, 410, 801 MPI_KEYVAL_INVALID, 333, 334, 335, 797 MPI_LAND, 206, 208, 802 MPI_LASTUSEDCODE, 429, 804 MPI_LB, 22, 728, 948 MPI_LOCK_EXCLUSIVE, 551, 797

MPI_LOCK_SHARED, 551, 552, 797 MPI_LOGICAL, 29, 207, 642, 800 MPI_LONG, 30, 207, 642, 799 MPI_LONG_DOUBLE, <u>30</u>, 207, 642, 799 MPI_LONG_DOUBLE_INT, 210, 801 MPI_LONG_INT, 210, 211, 801 MPI_LONG_LONG, 30, 207, 799, 955 MPI_LONG_LONG_INT, 30, 207, 642, 799, 955 MPI_LOR, 206, 208, 802 MPI_LXOR, 206, 208, 802 MPI_MAX, 205-207, 226, 802 MPI_MAX_DATAREP_STRING, 19, 607, 645, 798 MPI_MAX_ERROR_STRING, 19, 425, 430, 798 MPI_MAX_FROM_GROUP_STRINGTAG, 325 MPI_MAX_FROM_GROUP_TAG, 308, 798 MPI_MAX_INFO_KEY, 19, 427, 435, 438, 439, 798 MPI_MAX_INFO_VAL, 19, 427, 435, 798 MPI_MAX_LIBRARY_VERSION_STRING, 19, 408, 798, 948 MPI_MAX_OBJECT_NAME, 19, 346-349, 798, 949.956 MPI_MAX_PORT_NAME, 19, 490, 798 MPI_MAX_PROCESSOR_NAME, 19, 411, 798, 957MPI_MAX_PSET_NAME_LEN, 461, 798 MPI_MAXLOC, 206, 209, 210, 213, 802 MPI_MESSAGE_NO_PROC, <u>80</u>, <u>82</u>, <u>83</u>, <u>91</u>, 797, 949 MPI_MESSAGE_NULL, 80, 82, 83, 802, 949 MPI_MESSAGE_PROC_NULL, 80 MPI_MIN, 206, 207, 802 "mpi_minimum_memory_alignment", 412, 511, 513, 943 MPI_MINLOC, 206, 209, 210, 213, 802 MPI_MODE_APPEND, 594, 595, 805 MPI_MODE_CREATE, 594, 595, 604, 805 MPI_MODE_DELETE_ON_CLOSE, 594-596, 805 MPI_MODE_EXCL, 594, 595, 805 MPI_MODE_NOCHECK, 552, 557, 558, 805 MPI_MODE_NOPRECEDE, 546, 557, 558, 805 MPI_MODE_NOPUT, 557, 805 MPI_MODE_NOSTORE, 557, 805 MPI_MODE_NOSUCCEED, 557, 558, 805 MPI_MODE_RDONLY, 594, 595, 600, 805 MPI_MODE_RDWR, 594, 595, 805 MPI_MODE_SEQUENTIAL, 594, 595, 598, 605, 610, 616, 628, 651, 805, 958 MPI_MODE_UNIQUE_OPEN, 594, 595, 805 MPI_MODE_WRONLY, 594, 595, 805 MPI_NO_OP, 508, 532, 534, 802, 945 MPI_NULL_COPY_FN, 22, 332, 722, 804

MPI_NULL_DELETE_FN, 22, 333, <u>722</u> , 804	1
MPI_OFFSET, <u>31</u> , 207, 642, 799, 800, 952–955	2
MPI_OFFSET_KIND, 19, <u>20</u> , 31, 653, 758, 798	3
MPI_OP_NULL, 216, 802	
MPI_ORDER_C, 18, 122, 125, 126, 806	4
	5
MPI_ORDER_FORTRAN, 18, 122, 125, 806	6
MPI_PACKED, 15, 29, 30, 40, 41, 160, 162,	7
166,641,642,792,799,800	
MPI_PROC_NULL, 28, 32, 34, 77, 80, 82, 83,	8
91, 176, 178, 180, 182, 190, 192, 206,	9
281, 376, 380, 409, 410, 513, 514, 523,	10
797, 949, 955, 958	11
MPI_PROD, 206, 208, 802	
MPI_REAL, <u>29</u> , 40, 207, 641, 642, 749, 750,	12
	13
756, 800	14
MPI_REAL16, <u>29</u> , 207, 643, 800	15
MPI_REAL2, <u>29</u> , 207, 643, 800	
MPI_REAL4, <u>29</u> , 207, 643, 749, 754, 800	16
MPI_REAL8, 29, 207, 643, 749, 800, 953	17
MPI_REPLACE, 530-532, 534, 575, 802, 954,	18
958	19
MPI_REQUEST_NULL, 62-65, 68-72, 584, 802	
MPI_ROOT, 176, 797	20
MPI_SEEK_CUR, 622, 629, 806	21
MPI_SEEK_END, 622, 629, 806	22
	23
MPI_SEEK_SET, 622, 623, 629, 806	
MPI_SESSION_NULL, 458, 802	24
"mpi_shared_memory", 304, 942	25
MPI_SHORT, <u>30</u> , 207, 642, 799	26
MPI_SHORT_INT, 210, 801	27
MPI_SIGNED_CHAR, <u>30</u> , 207, 209, 642, 799,	28
955	
MPI_SIMILAR, <u>282</u> , 291, 321, 801	29
"mpi_size", 458, 462, 466	30
MPI_SOURCE, 34, <u>35</u> , 228, 782, 798, 944, 951	31
MPI_STATUS_IGNORE, 10, 19, <u>37</u> , 583, 610,	32
736, 758, 782–784, 792, 806, 807, 948	
	33
MPI_STATUS_SIZE, 19, <u>35</u> , 738, 798, 951	34
MPI_STATUSES_IGNORE, 18, 19, <u>37</u> , 583, 585,	35
758, 782–784, 806, 807	36
MPI_SUBARRAYS_SUPPORTED, 19, 732, 733,	
736-740, 744-747, 759-761, 798, 950	37
MPI_SUBVERSION, 19, 408, 807	38
MPI_SUCCESS, 22, 23, 62, 70, 72, 332–334,	39
336, 337, 340, 341, 357, 425-427,	40
431-433, 445, 481, 647, 672, 681, 690,	41
692–694, 700, 704, 705, 709, 716, 718,	
720, 722, 795, 943, 944	42
MPI_SUM, 206, 208, 530, 791, 802	43
	44
MPI_T_BIND_MPI_COMM, <u>673</u> , 808	45
MPI_T_BIND_MPI_DATATYPE, <u>673</u> , 808	46
MPI_T_BIND_MPI_ERRHANDLER, <u>673</u> , 808	
MPI_T_BIND_MPI_FILE, <u>673</u> , 808	47
MPI_T_BIND_MPI_GROUP, <u>673</u> , 808	48

1	
-	MPI_T_BIND_MPI_INFO, <u>673</u> , 808
2	MPI_T_BIND_MPI_MESSAGE, 673, 808
3	MPI_T_BIND_MPI_OP, <u>673</u> , 808
	MPI_T_BIND_MPI_REQUEST, <u>673</u> , 808
4	MPI_T_BIND_MPI_SESSION, 673
5	MPI_T_BIND_MPI_WIN, <u>673</u> , 808
6	
7	MPI_T_BIND_NO_OBJECT, <u>673</u> , 680, 682,
8	689, 691, 704, 705, 808
	MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE,
9	700, <u>701</u> , 809
10	MPI_T_CB_REQUIRE_MPI_RESTRICTED,
11	$\underline{700}, 700, \underline{701}, 809$
12	MPI_T_CB_REQUIRE_NONE, 700, 809
13	MPI_T_CB_REQUIRE_THREAD_SAFE, 700,
	<u>701</u> , 809
14	MPI_T_CVAR_HANDLE_NULL, <u>683</u> , 807
15	MPI_T_ENUM_NULL, 679, 689, 704, 807
16	MPI_T_ERR_CANNOT_INIT, 720, 797
17	MPI_T_ERR_CVAR_SET_NEVER, 684, 720,
18	797
	MPI_T_ERR_CVAR_SET_NOT_NOW, <u>684</u> ,
19	720, 797
20	MPI_T_ERR_INVALID, 720, 797, 946
21	MPI_T_ERR_INVALID_HANDLE, 691, 709,
22	720, 797
23	MPI_T_ERR_INVALID_INDEX, 22, 700, 704,
24	$\frac{720}{725}, 725, 797, 943$
25	MPI_T_ERR_INVALID_ITEM, 22, <u>720</u> , 725,
20	
26	797, 943 MDL T EDD INVALID NAME 681 600 705
26 27	MPI_T_ERR_INVALID_NAME, 681, 690, 705,
	MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, 720, 797, 946
27	MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, <u>720</u> , 797, 946 MPI_T_ERR_INVALID_SESSION, <u>720</u> , 797
27 28 29	MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, <u>720</u> , 797, 946 MPI_T_ERR_INVALID_SESSION, <u>720</u> , 797 MPI_T_ERR_MEMORY, <u>720</u> , 797
27 28 29 30	MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, <u>720</u> , 797, 946 MPI_T_ERR_INVALID_SESSION, <u>720</u> , 797 MPI_T_ERR_MEMORY, <u>720</u> , 797 MPI_T_ERR_NOT_ACCESSIBLE, <u>701</u> , 702,
27 28 29 30 31	MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, <u>720</u> , 797, 946 MPI_T_ERR_INVALID_SESSION, <u>720</u> , 797 MPI_T_ERR_MEMORY, <u>720</u> , 797 MPI_T_ERR_NOT_ACCESSIBLE, <u>701</u> , 702, 720, 797
27 28 29 30	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, <u>720</u>, 797, 946 MPI_T_ERR_INVALID_SESSION, <u>720</u>, 797 MPI_T_ERR_MEMORY, <u>720</u>, 797 MPI_T_ERR_NOT_ACCESSIBLE, <u>701</u>, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, <u>720</u>, 797
27 28 29 30 31	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, <u>720</u>, 797, 946 MPI_T_ERR_INVALID_SESSION, <u>720</u>, 797 MPI_T_ERR_MEMORY, <u>720</u>, 797 MPI_T_ERR_NOT_ACCESSIBLE, <u>701</u>, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, <u>720</u>, 797 MPI_T_ERR_NOT_SUPPORTED, <u>700</u>, 720,
27 28 29 30 31 32	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, <u>720</u>, 797, 946 MPI_T_ERR_INVALID_SESSION, <u>720</u>, 797 MPI_T_ERR_MEMORY, <u>720</u>, 797 MPI_T_ERR_NOT_ACCESSIBLE, <u>701</u>, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, <u>720</u>, 797 MPI_T_ERR_NOT_SUPPORTED, <u>700</u>, 720, 797
27 28 29 30 31 32 33	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, <u>720</u>, 797, 946 MPI_T_ERR_INVALID_SESSION, <u>720</u>, 797 MPI_T_ERR_MEMORY, <u>720</u>, 797 MPI_T_ERR_NOT_ACCESSIBLE, <u>701</u>, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, <u>720</u>, 797 MPI_T_ERR_NOT_SUPPORTED, <u>700</u>, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, <u>720</u>, 797
27 28 29 30 31 32 33 34 35	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, <u>720</u>, 797, 946 MPI_T_ERR_INVALID_SESSION, <u>720</u>, 797 MPI_T_ERR_MEMORY, <u>720</u>, 797 MPI_T_ERR_NOT_ACCESSIBLE, <u>701</u>, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, <u>720</u>, 797 MPI_T_ERR_NOT_SUPPORTED, <u>700</u>, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, <u>720</u>, 797 MPI_T_ERR_OUT_OF_SESSIONS, <u>720</u>, 797
27 28 29 30 31 32 33 34 35 36	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, <u>720</u>, 797, 946 MPI_T_ERR_INVALID_SESSION, <u>720</u>, 797 MPI_T_ERR_MEMORY, <u>720</u>, 797 MPI_T_ERR_NOT_ACCESSIBLE, <u>701</u>, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, <u>720</u>, 797 MPI_T_ERR_NOT_SUPPORTED, <u>700</u>, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, <u>720</u>, 797
27 28 29 30 31 32 33 34 35 36 37	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, <u>720</u>, 797, 946 MPI_T_ERR_INVALID_SESSION, <u>720</u>, 797 MPI_T_ERR_MEMORY, <u>720</u>, 797 MPI_T_ERR_NOT_ACCESSIBLE, <u>701</u>, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, <u>720</u>, 797 MPI_T_ERR_NOT_SUPPORTED, <u>700</u>, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, <u>720</u>, 797 MPI_T_ERR_OUT_OF_SESSIONS, <u>720</u>, 797 MPI_T_ERR_PVAR_NO_ATOMIC, <u>694</u>, 720, 797
27 28 29 30 31 32 33 34 35 36	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, 720, 797, 946 MPI_T_ERR_INVALID_SESSION, 720, 797 MPI_T_ERR_MEMORY, 720, 797 MPI_T_ERR_NOT_ACCESSIBLE, 701, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, 720, 797 MPI_T_ERR_NOT_SUPPORTED, 700, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, 720, 797 MPI_T_ERR_OUT_OF_SESSIONS, 720, 797 MPI_T_ERR_PVAR_NO_ATOMIC, 694, 720,
27 28 29 30 31 32 33 34 35 36 37	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, <u>720</u>, 797, 946 MPI_T_ERR_INVALID_SESSION, <u>720</u>, 797 MPI_T_ERR_MEMORY, <u>720</u>, 797 MPI_T_ERR_NOT_ACCESSIBLE, <u>701</u>, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, <u>720</u>, 797 MPI_T_ERR_NOT_SUPPORTED, <u>700</u>, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, <u>720</u>, 797 MPI_T_ERR_OUT_OF_SESSIONS, <u>720</u>, 797 MPI_T_ERR_PVAR_NO_ATOMIC, <u>694</u>, 720, 797
27 28 29 30 31 32 33 34 35 36 37 38	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, <u>720</u>, 797, 946 MPI_T_ERR_INVALID_SESSION, <u>720</u>, 797 MPI_T_ERR_MEMORY, <u>720</u>, 797 MPI_T_ERR_NOT_ACCESSIBLE, <u>701</u>, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, <u>720</u>, 797 MPI_T_ERR_NOT_SUPPORTED, <u>700</u>, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, <u>720</u>, 797 MPI_T_ERR_OUT_OF_SESSIONS, <u>720</u>, 797 MPI_T_ERR_PVAR_NO_ATOMIC, <u>694</u>, 720, 797 MPI_T_ERR_PVAR_NO_STARTSTOP, <u>692</u>,
27 28 29 30 31 32 33 34 35 36 37 38 39	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, <u>720</u>, 797, 946 MPI_T_ERR_INVALID_SESSION, <u>720</u>, 797 MPI_T_ERR_MEMORY, <u>720</u>, 797 MPI_T_ERR_NOT_ACCESSIBLE, <u>701</u>, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, <u>720</u>, 797 MPI_T_ERR_NOT_SUPPORTED, <u>700</u>, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, <u>720</u>, 797 MPI_T_ERR_OUT_OF_SESSIONS, <u>720</u>, 797 MPI_T_ERR_PVAR_NO_ATOMIC, <u>694</u>, 720, 797 MPI_T_ERR_PVAR_NO_STARTSTOP, <u>692</u>, 693, 720, 797
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, 720, 797, 946 MPI_T_ERR_INVALID_SESSION, 720, 797 MPI_T_ERR_MEMORY, 720, 797 MPI_T_ERR_NOT_ACCESSIBLE, 701, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, 720, 797 MPI_T_ERR_NOT_SUPPORTED, 700, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, 720, 797 MPI_T_ERR_OUT_OF_SESSIONS, 720, 797 MPI_T_ERR_PVAR_NO_ATOMIC, 694, 720, 797 MPI_T_ERR_PVAR_NO_STARTSTOP, 692, 693, 720, 797 MPI_T_ERR_PVAR_NO_WRITE, 694, 720, 797
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, 720, 797, 946 MPI_T_ERR_INVALID_SESSION, 720, 797 MPI_T_ERR_MEMORY, 720, 797 MPI_T_ERR_NOT_ACCESSIBLE, 701, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, 720, 797 MPI_T_ERR_NOT_SUPPORTED, 700, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, 720, 797 MPI_T_ERR_OUT_OF_SESSIONS, 720, 797 MPI_T_ERR_PVAR_NO_ATOMIC, 694, 720, 797 MPI_T_ERR_PVAR_NO_STARTSTOP, 692, 693, 720, 797 MPI_T_ERR_PVAR_NO_WRITE, 694, 720, 797 MPI_T_ERR_PVAR_NO_WRITE, 694, 720, 797 MPI_T_PVAR_ALL_HANDLES, 692, 693-695,
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, 720, 797, 946 MPI_T_ERR_INVALID_SESSION, 720, 797 MPI_T_ERR_MEMORY, 720, 797 MPI_T_ERR_NOT_ACCESSIBLE, 701, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, 720, 797 MPI_T_ERR_NOT_SUPPORTED, 700, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, 720, 797 MPI_T_ERR_OUT_OF_SESSIONS, 720, 797 MPI_T_ERR_PVAR_NO_ATOMIC, 694, 720, 797 MPI_T_ERR_PVAR_NO_STARTSTOP, 692, 693, 720, 797 MPI_T_ERR_PVAR_NO_WRITE, 694, 720, 797 MPI_T_PVAR_ALL_HANDLES, 692, 693-695, 808
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, 720, 797, 946 MPI_T_ERR_INVALID_SESSION, 720, 797 MPI_T_ERR_MEMORY, 720, 797 MPI_T_ERR_NOT_ACCESSIBLE, 701, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, 720, 797 MPI_T_ERR_NOT_SUPPORTED, 700, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, 720, 797 MPI_T_ERR_OUT_OF_SESSIONS, 720, 797 MPI_T_ERR_PVAR_NO_ATOMIC, 694, 720, 797 MPI_T_ERR_PVAR_NO_STARTSTOP, 692, 693, 720, 797 MPI_T_ERR_PVAR_NO_WRITE, 694, 720, 797 MPI_T_PVAR_ALL_HANDLES, 692, 693-695, 808 MPI_T_PVAR_CLASS_AGGREGATE, 686, 687, 808
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, 720, 797, 946 MPI_T_ERR_INVALID_SESSION, 720, 797 MPI_T_ERR_MEMORY, 720, 797 MPI_T_ERR_NOT_ACCESSIBLE, 701, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, 720, 797 MPI_T_ERR_NOT_SUPPORTED, 700, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, 720, 797 MPI_T_ERR_OUT_OF_SESSIONS, 720, 797 MPI_T_ERR_PVAR_NO_ATOMIC, 694, 720, 797 MPI_T_ERR_PVAR_NO_STARTSTOP, 692, 693, 720, 797 MPI_T_ERR_PVAR_NO_WRITE, 694, 720, 797 MPI_T_PVAR_ALL_HANDLES, 692, 693-695, 808 MPI_T_PVAR_CLASS_AGGREGATE, 686, 687, 808 MPI_T_PVAR_CLASS_COUNTER, 696, 808
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, 720, 797, 946 MPI_T_ERR_INVALID_SESSION, 720, 797 MPI_T_ERR_MEMORY, 720, 797 MPI_T_ERR_NOT_ACCESSIBLE, 701, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, 720, 797 MPI_T_ERR_NOT_SUPPORTED, 700, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, 720, 797 MPI_T_ERR_OUT_OF_SESSIONS, 720, 797 MPI_T_ERR_PVAR_NO_ATOMIC, 694, 720, 797 MPI_T_ERR_PVAR_NO_STARTSTOP, 692, 693, 720, 797 MPI_T_ERR_PVAR_NO_WRITE, 694, 720, 797 MPI_T_PVAR_ALL_HANDLES, 692, 693-695, 808 MPI_T_PVAR_CLASS_AGGREGATE, 686, 687, 808 MPI_T_PVAR_CLASS_COUNTER, 686, 808 MPI_T_PVAR_CLASS_GENERIC, 687, 808
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, 720, 797, 946 MPI_T_ERR_INVALID_SESSION, 720, 797 MPI_T_ERR_MEMORY, 720, 797 MPI_T_ERR_NOT_ACCESSIBLE, 701, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, 720, 797 MPI_T_ERR_NOT_SUPPORTED, 700, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, 720, 797 MPI_T_ERR_OUT_OF_SESSIONS, 720, 797 MPI_T_ERR_PVAR_NO_ATOMIC, 694, 720, 797 MPI_T_ERR_PVAR_NO_STARTSTOP, 692, 693, 720, 797 MPI_T_ERR_PVAR_NO_WRITE, 694, 720, 797 MPI_T_PVAR_ALL_HANDLES, 692, 693-695, 808 MPI_T_PVAR_CLASS_AGGREGATE, 686, 687, 808 MPI_T_PVAR_CLASS_GENERIC, 687, 808 MPI_T_PVAR_CLASS_HIGHWATERMARK,
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	 MPI_T_ERR_INVALID_NAME, 681, 690, 705, 716, 720, 797, 946 MPI_T_ERR_INVALID_SESSION, 720, 797 MPI_T_ERR_MEMORY, 720, 797 MPI_T_ERR_NOT_ACCESSIBLE, 701, 702, 720, 797 MPI_T_ERR_NOT_INITIALIZED, 720, 797 MPI_T_ERR_NOT_SUPPORTED, 700, 720, 797 MPI_T_ERR_OUT_OF_HANDLES, 720, 797 MPI_T_ERR_OUT_OF_SESSIONS, 720, 797 MPI_T_ERR_PVAR_NO_ATOMIC, 694, 720, 797 MPI_T_ERR_PVAR_NO_STARTSTOP, 692, 693, 720, 797 MPI_T_ERR_PVAR_NO_WRITE, 694, 720, 797 MPI_T_PVAR_ALL_HANDLES, 692, 693-695, 808 MPI_T_PVAR_CLASS_AGGREGATE, 686, 687, 808 MPI_T_PVAR_CLASS_COUNTER, 686, 808 MPI_T_PVAR_CLASS_GENERIC, 687, 808

MPI_T_PVAR_CLASS_LOWWATERMARK, <u>686</u>, 808 MPI_T_PVAR_CLASS_PERCENTAGE, 686, 808 MPI_T_PVAR_CLASS_SIZE, 686, 808 MPI_T_PVAR_CLASS_STATE, 685, 808 MPI_T_PVAR_CLASS_TIMER, <u>687</u>, 808 MPI_T_PVAR_HANDLE_NULL, 692, 807 MPI_T_PVAR_SESSION_NULL, 690, 807 MPI_T_SCOPE_ALL, 680, 808 MPI_T_SCOPE_ALL_EQ, 680, 684, 808 MPI_T_SCOPE_CONSTANT, 680, 808 MPI_T_SCOPE_GROUP, 680, 808 MPI_T_SCOPE_GROUP_EQ, <u>680</u>, 684, 808 MPI_T_SCOPE_LOCAL, 680, 808 MPI_T_SCOPE_READONLY, 680, 808 MPI_T_SOURCE_ORDERED, 699, 809 MPI_T_SOURCE_UNORDERED, 699, 809 MPI_T_VERBOSITY_MPIDEV_ALL, 672, 673, 807 MPI_T_VERBOSITY_MPIDEV_BASIC, 673, 807 MPI_T_VERBOSITY_MPIDEV_DETAIL, 673, 807 MPI_T_VERBOSITY_TUNER_ALL, 673, 807 MPI_T_VERBOSITY_TUNER_BASIC, 673, 807 MPI_T_VERBOSITY_TUNER_DETAIL, 673, 807 MPI_T_VERBOSITY_USER_ALL, 673, 807 MPI_T_VERBOSITY_USER_BASIC, 672, 673, 807 MPI_T_VERBOSITY_USER_DETAIL, 672, 673, 807 MPI_TAG, 34, <u>35</u>, 228, 782, 798, 944, 951 MPI_TAG_UB, 31, 409, 787, 791, 801 MPI_THREAD_FUNNELED, 447, 448, 805 "MPI_THREAD_FUNNELED", 457 MPI_THREAD_MULTIPLE, 447, 448, 450, 805, 946 "MPI_THREAD_MULTIPLE", 457 MPI_THREAD_SERIALIZED, 447, 448, 805 "MPI_THREAD_SERIALIZED", 457 MPI_THREAD_SINGLE, 447-449, 457, 805 "MPI_THREAD_SINGLE", 457 "mpi_thread_support_level", 457 MPI_TYPE_DUP_FN, <u>340</u>, 803 MPI_TYPE_NULL_COPY_FN, 340, 803 MPI_TYPE_NULL_DELETE_FN, 340, 803, 951 MPI_TYPECLASS_COMPLEX, 755, 806 MPI_TYPECLASS_INTEGER, 755, 806 MPI_TYPECLASS_REAL, 755, 806 MPI_UB, 4, 22, 728, 948 MPI_UINT16_T, 30, 207, 642, 799, 952-955 MPI_UINT32_T, <u>30</u>, 207, 642, 676, 799, 952-955

MPI_UINT64_T, <u>30</u>, 207, 642, 676, 799, 952-955 MPI_UINT8_T, 30, 207, 642, 799, 952-955 MPI_UNDEFINED, 36, 68, 69, 72, 73, 132, 135, 137, 142, 163, 280, 281, 299, 300, 303, 367, 378, 379, 751, 797, 948, 955 MPI_UNEQUAL, 282, 291, 320, 801 MPI_UNIVERSE_SIZE, 476, 498, 499, 804 MPI_UNSIGNED, 30, 207, 642, 676, 685–687, 799 MPI_UNSIGNED_CHAR, <u>30</u>, 207, 209, 642, 799 MPI_UNSIGNED_LONG, 30, 207, 642, 676, 686, 687, 799 MPI_UNSIGNED_LONG_LONG, 30, 207, 642, 676, 686, 687, 799, 955 MPI_UNSIGNED_SHORT, 30, 207, 642, 799 MPI_UNWEIGHTED, 19, 362, 364-366, 373, 375, 758, 806, 948, 954 MPI_VAL, 16, 780 MPI_VERSION, 19, 408, 807 MPI_WCHAR, 30, 209, 349, 641, 642, 799, 955 MPI_WEIGHTS_EMPTY, 19, 362, 364, 758, 806, 948 MPI_WIN_BASE, 519, 791, 804 MPI_WIN_CREATE_FLAVOR, 519, 804 MPI_WIN_DISP_UNIT, 519, 804 MPI_WIN_DUP_FN, 336, 803 MPI_WIN_FLAVOR_ALLOCATE, 519, 804 MPI_WIN_FLAVOR_CREATE, 519, 804 MPI_WIN_FLAVOR_DYNAMIC, 520, 804 MPI_WIN_FLAVOR_SHARED, 514, 520, 804 MPI_WIN_MODEL, 519, 541, 804 MPI_WIN_NULL, 518, 802 MPI_WIN_NULL_COPY_FN, 336, 803 MPI_WIN_NULL_DELETE_FN, 337, 803 MPI_WIN_SEPARATE, 520, 541, 561, 804 MPI_WIN_SIZE, 519, 804 MPI_WIN_UNIFIED, 520, 541, 561, 570, 804 MPI_WTIME_IS_GLOBAL, 409, 410, 434, 787, 801"mpix://UNIVERSE", 458 native, 638 "nb_proc", 604 "no_locks", 508, 519 "none", 568 "num_io_nodes", 604 "path", 485 "random", 603 "rar", 568 "raw", 568

"striping_factor", 604 "striping_unit", 604 "thread_level", 446 "true", 309, 519, 603 "war", 568 "waw", 568 "wdir", 446, 485 "write_mostly", 603 "write_once", 603

"reverse_sequential", 603

"same_op_no_op", 508

"same_size", 508, 945

"soft", 446, 480, 485

"same_op", 508

"sequential", 603

"same_disp_unit", 508, 946

"read_mostly", 603 "read_once", 603

1

 $\mathbf{2}$

3

4

5

6

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43 44

45

46

47

MPI Declarations Index

10

11 12 This index refers to declarations needed in C and Fortran, such as address kind integers, handles, etc. The underlined page numbers is the "main" reference (sometimes there are more than one when key concepts are discussed in multiple areas). Fortran defined types are shown as TYPE(*name*).

13	MPI_Aint, <u>20</u> , 20, 31, <u>111</u> , 114, 116, 119,
14	129-131, 134-136, 145, 166-168, 507,
15	509, 511, 515, 516, 519, 523, 525, 529,
16	531, 533-535, 537-539, 640, 645, 699,
17	758, 788, 799, 809
	MPI_Comm, 16, <u>28</u> , 282, 289, 290, 292–294,
18	299, 302, 308310, 320323, 325, 331,
19	334, 335, 460, 801, 802, 809, 810
20	TYPE(MPI_Comm), 745, 801, 802, 810
21	MPI_Count, <u>20</u> , 31, 698, 699, 799, 809, 948
22	MPI_Datatype, <u>111</u> , 340, 767, 799-802, 809, 810
23	TYPE(MPI_Datatype), 799-802, 810
24	MPI_Errhandler, <u>417</u> , 418–424, 781, 798, 802,
25	809, 810
	TYPE(MPI_Errhandler), 798, 802, 810
26	MPI_F08_status, <u>783</u> , 807, 809, 951
27	MPI_File, 421, 422, <u>593</u> , 596–599, 601, 602,
28	604,606,610-629,631-637,640,
29	649-651, 781, 802, 809, 810
30	TYPE(MPI_File), 802, 810
31	MPI_Fint, <u>780</u> , 807, 809, 955
32	MPI_Group, <u>280</u> , 281–286, 288, 321, 460, 520,
33	547, 548, 599, 781, 802, 809, 810
34	TYPE(MPI_Group), 802, 810
	MPI_Info, 411, <u>435</u> , 436–441, 458, 462, 477,
35	480, 482, 489, 493–495, 502, 521, 593,
36	596, 601, 602, 604, 724, 781, 801, 802,
37	809, 810, 957
38	TYPE(MPI_Info), 801, 802, 810
39	MPI_Message, <u>79</u> , 781, 797, 802, 809, 810, 949 TYPE(MPI_Message), 797, 802, 810
40	MPI_Offset, <u>20</u> , 20, 31, 597–599, 604, 606,
41	610–615, 622, 623, 628, 629, 631, 632,
42	645, 653, 699, 779, 799, 809
43	MPI_Op, 204, <u>213</u> , 216, 218–222, 224–226,
	241-246, 260-265, 529, 531, 533, 538,
44	539, 781, 802, 809, 810
45	TYPE(MPI_Op), 802, 810
46	MPI_Request, 56–61, <u>63</u> , 65, 67–71, 73, 75, 84,
47	
	86-90, 96-100, 425, 582, 585,

613-615, 619-621, 625, 626, 760, 781, 802, 809, 810 TYPE(MPI_Request), 802, 810 MPI_Session, 423, 781, 802, 809, 810 TYPE(MPI_Session), 810 MPI_Status, <u>32</u>, 35, 37–39, 63, 67–71, 73, 75-77, 79, 81, 85, 141, 583, 588, 589, 610-612, 616-618, 624, 627, 632-637, 735, 782-785, 806, 809, 810, 944, 948, 951 TYPE(MPI_Status), 35, 745, 783-785, 806, 810, 941, 944, 951 MPI_T_cb_safety, 809 MPI_T_cvar_handle, <u>682</u>, <u>683</u>, <u>807</u> MPI_T_enum, 676, 677, 678, 688, 702, 807 MPI_T_event_instance, 708 MPI_T_event_registration, 705 MPI_T_pvar_handle, <u>691</u>, 692-694, 807, 808 MPI_T_pvar_session, 690, 691-694, 807 MPI_T_source_order, 809 MPI_Win, 337-339, 419, 420, 507, 509, 511, 515, 518, 520, 521, 523, 525, 529, 531, 533-535, 537-539, 544, 547-549, 551-556, 781, 802, 809, 810 TYPE(MPI_Win), 802, 810

MPI Callback Function Prototype Index

This index lists the C typedef names for callback routines, such as those used with attribute caching or user-defined reduction operations. Fortran example prototypes are given near the text of the C name.

MPI_Comm_copy_attr_function, 22, <u>331</u> , 332,	15
734, 803, 810	16
MPI_Comm_delete_attr_function, 22, <u>331</u> , 332,	17
803, 810	18
MPI_Comm_errhandler_fn, 724, 954	19
MPI_Comm_errhandler_function, 22, 417, 724,	20
728, 811, 954	
MPI_Copy_function, 22, 721, 804, 816	21
MPI_Datarep_conversion_function, 645, 803, 811	22
MPI_Datarep_extent_function, 645, 811	23
MPI_Delete_function, 22, 722, 804, 816	24
MPI_File_errhandler_fn, 724, 954	25
MPI_File_errhandler_function, 421, 724, 811,	26
954 MDL Creativest connect function 581 911	27
MPI_Grequest_cancel_function, 584, 811 MPI_Grequest_free_function, 583, 811	28
MP1_Grequest_query_function, 583, 811 MPI_Grequest_query_function, 583, 811	29
MPI_Handler_function, 22, 728, 948	30
MPI_Session_errhandler_function, 423, 811	31
MPI_T_event_cb_function, 707, 811	
MPI_T_event_dropped_function, 710, 811	32
MPI_T_event_free_cb_function, 709, 811	33
MPI_Type_copy_attr_function, 339, 340, 803,	34
810	35
MPI_Type_delete_attr_function, 339, 340, 803,	36
810, 951	37
MPI_User_function, 214, 217, 810	38
MPI_Win_copy_attr_function, 336, 337, 803,	39
810	40
MPI_Win_delete_attr_function, 336, 337, 803,	41
810	
$MPI_Win_errhandler_fn, 724, 954$	42
MPI_Win_errhandler_function, 419, 724, 811,	43
954	44
	45
	46
	47

MPI Function Index

10

The underlined page numbers refer to the function definitions.

11MPI ABORT. 469 12MPI_ACCUMULATE, 529 MPI_ADD_ERROR_CLASS, 429 13MPI_ADD_ERROR_CODE, 430 14MPI_ADD_ERROR_STRING, 430 15MPI_AINT_ADD, 130 16MPI_AINT_DIFF, 130 17MPI_ALLGATHER, 195 18MPI_ALLGATHER_INIT, 254 19MPI_ALLGATHERV, 197 MPI_ALLGATHERV_INIT, 255 20MPI_ALLOC_MEM, 411 21MPI_ALLREDUCE, 218 22MPI_ALLREDUCE_INIT, 261 23MPI_ALLTOALL, 198 24 MPI_ALLTOALL_INIT, 256 25MPI_ALLTOALLV, 200 26MPI ALLTOALLV INIT. 257 27MPI_ALLTOALLW, 202 MPI_ALLTOALLW_INIT, 259 28MPI_ATTR_DELETE, 723 29MPI_ATTR_GET, 723, 788 30 MPI_ATTR_PUT, 723, 788, 791 31 MPI_BARRIER, 177 32 MPI_BARRIER_INIT, 248 33 MPI_BCAST, <u>178</u>, 269 34MPI_BCAST_INIT, 249 MPI_BSEND, 45 35MPI_BSEND_INIT, 87 36 MPI_BUFFER_ATTACH, 52 37 MPI_BUFFER_DETACH, 52 38 MPI_CANCEL, <u>84</u>, 452 39MPI_CART_COORDS, 370 40MPI_CART_CREATE, 356 41MPI_CART_GET, 369 42MPI_CART_MAP, 378 MPI_CART_RANK, 370 43MPI_CART_SHIFT, 375 44MPI_CART_SUB, <u>377</u> 45MPI_CARTDIM_GET, 368 46MPI_CLOSE_PORT, 490 47 MPI_COMM_ACCEPT, 491 48

MPI COMM C2F. 780 MPI_COMM_CALL_ERRHANDLER, 431 MPI_COMM_COMPARE, 291 MPI_COMM_CONNECT, 492 MPI_COMM_CREATE, 295 MPI_COMM_CREATE_ERRHANDLER, 416, 417.813 MPI_COMM_CREATE_FROM_GROUP, 306 MPI_COMM_CREATE_GROUP, 297 MPI_COMM_CREATE_KEYVAL, 329, 330, 343, 787, 812, 956 MPI_COMM_DELETE_ATTR, 329, 335, 343 MPI_COMM_DISCONNECT, 501 MPI_COMM_DUP, 292 MPI_COMM_DUP_FN, <u>332</u>, 946 MPI_COMM_DUP_WITH_INFO, 293 MPI_COMM_F2C, 780 MPI_COMM_FREE, 308, 451, 458 MPI_COMM_FREE_KEYVAL, 329, 333, 343 MPI_COMM_GET_ATTR, 329, 334, 343, 740, 788, 791 MPI_COMM_GET_ERRHANDLER, 416, 418, 957 MPI_COMM_GET_INFO, 310 MPI_COMM_GET_NAME, 346 MPI_COMM_GET_PARENT, 481 MPI_COMM_GROUP, 282 MPI_COMM_IDUP, 293 MPI_COMM_IDUP_WITH_INFO, 294 MPI_COMM_JOIN, 502 MPI_COMM_NULL_COPY_FN, 332, 946 MPI_COMM_NULL_DELETE_FN, <u>332</u>, 946 MPI_COMM_RANK, 290, 741 MPI_COMM_RANK_F08, 741 MPI_COMM_REMOTE_GROUP, 321 MPI_COMM_REMOTE_SIZE, 321 MPI_COMM_SET_ATTR, 329, 334, 343, 740, 788, 791 MPI_COMM_SET_ERRHANDLER, 416, 418 MPI_COMM_SET_INFO, 310 MPI_COMM_SET_NAME, 345 MPI_COMM_SIZE, 289

MPI_COMM_SPAWN, 477 MPI_COMM_SPAWN_MULTIPLE, <u>482</u> MPI_COMM_SPLIT, 299 MPI_COMM_SPLIT_TYPE, 302 MPI_COMM_TEST_INTER, 320 MPI_COMPARE_AND_SWAP, 534 MPI_CONVERSION_FN_NULL, 647 MPI_DIMS_CREATE, 357 MPI_DIST_GRAPH_CREATE, 363 MPI_DIST_GRAPH_CREATE_ADJACENT, 361MPI_DIST_GRAPH_NEIGHBORS, 374 MPI_DIST_GRAPH_NEIGHBORS_COUNT, <u>373</u> MPI_DUP_FN, 722 MPI_ERRHANDLER_C2F, 781 MPI_ERRHANDLER_F2C, 781 MPI_ERRHANDLER_FREE, <u>424</u>, 451, 458 MPI_ERROR_CLASS, 426MPI_ERROR_STRING, 425MPI_EXSCAN, 225 MPI_EXSCAN_INIT, 265 MPI_F_SYNC_REG, 748 MPI_FETCH_AND_OP, 533 MPI_FILE_C2F, <u>781</u> MPI_FILE_CALL_ERRHANDLER, <u>432</u> MPI_FILE_CLOSE, 596 MPI_FILE_CREATE_ERRHANDLER, 416, 421, 813 MPI_FILE_DELETE, 596 MPI_FILE_F2C, 781 MPI_FILE_GET_AMODE, 600 MPI_FILE_GET_ATOMICITY, 650 MPI_FILE_GET_BYTE_OFFSET, 623 MPI_FILE_GET_ERRHANDLER, 416, 422, 957 MPI_FILE_GET_GROUP, 599 MPI_FILE_GET_INFO, 602 MPI_FILE_GET_POSITION, 622 MPI_FILE_GET_POSITION_SHARED, 629 MPI_FILE_GET_SIZE, 599 MPI_FILE_GET_TYPE_EXTENT, 640 MPI_FILE_GET_VIEW, 606 MPI_FILE_IREAD, 619 MPI_FILE_IREAD_ALL, 620 MPI_FILE_IREAD_AT, 613 MPI_FILE_IREAD_AT_ALL, 613 MPI_FILE_IREAD_SHARED, 625 MPI_FILE_IWRITE, 621 MPI_FILE_IWRITE_ALL, 621 MPI_FILE_IWRITE_AT, 614 MPI_FILE_IWRITE_AT_ALL, 615 MPI_FILE_IWRITE_SHARED, 626 MPI_FILE_OPEN, 593

MPI_FILE_PREALLOCATE, <u>598</u>	1
MPI_FILE_READ, <u>616</u>	2
MPI_FILE_READ_ALL, <u>617</u>	3
MPI_FILE_READ_ALL_BEGIN, <u>633</u> , 773	4
MPI_FILE_READ_ALL_END, <u>634</u> , 773	5
MPI_FILE_READ_ALL_FN, 14	
MPI_FILE_READ_AT, <u>610</u>	6
MPI_FILE_READ_AT_ALL, <u>611</u>	7
MPI_FILE_READ_AT_ALL_BEGIN, <u>631</u> , 773	8
MPI_FILE_READ_AT_ALL_END, <u>632</u> , 773	9
MPI_FILE_READ_AT_ALL_FN, 14	10
MPI_FILE_READ_ORDERED, 627	11
MPI_FILE_READ_ORDERED_BEGIN, 635,	
773	12
MPI_FILE_READ_ORDERED_END, <u>636</u> ,	13
773	14
MPI_FILE_READ_ORDERED_FN, 14	15
MPI_FILE_READ_SHARED, <u>624</u>	16
$MPI_FILE_SEEK, 622$	17
MPI_FILE_SEEK_SHARED, 628	18
MPI_FILE_SET_ATOMICITY, <u>650</u>	
MPI_FILE_SET_ERRHANDLER, 416, 421	19
$MPI_FILE_SET_INFO, 601$	20
MPI_FILE_SET_SIZE, <u>597</u>	21
MPI_FILE_SET_VIEW, <u>604</u>	22
$MPI_FILE_SYNC, 651$	23
$MPI_FILE_STRVC, 051$ $MPI_FILE_WRITE, 618$	24
MPI_FILE_WRITE_ALL, <u>618</u>	25
$MPI_FILE_WRITE_ALL_BEGIN, 634, 773$	
MPI_FILE_WRITE_ALL_END, <u>635</u> , 773	26
MPI_FILE_WRITE_ALL_FN, 14	27
$MPI_FILE_WRITE_AT, 611$	28
$MPI_FILE_WRITE_AT_ALL, 612$	29
$MPI_FILE_WRITE_AT_ALL_BEGIN, 632,$	30
773	31
MPI_FILE_WRITE_AT_ALL_END, <u>633</u> , 773	32
MPI_FILE_WRITE_AT_ALL_END, <u>055</u> , 775 MPI_FILE_WRITE_AT_ALL_FN, 14	
MPI_FILE_WRITE_ORDERED, <u>628</u>	33
MPI_FILE_WRITE_ORDERED_BEGIN,	34
636, 773	35
MPI_FILE_WRITE_ORDERED_END, <u>637</u> ,	36
773	37
MPI_FILE_WRITE_ORDERED_FN, 14	38
MPI_FILE_WRITE_SHARED, 625	39
MPI_FINALIZE, <u>450</u> , 451, 672, 682	40
MPI_FINALIZED, <u>450</u> , 451, 072, 082 MPI_FINALIZED, <u>454</u>	
·	41
$\begin{array}{l} \text{MPI_FREE_MEM, } \underline{413} \\ \text{MPI_CATHER_170} \end{array}$	42
MPI_GATHER, <u>179</u> MPI_CATHER_INIT_250	43
MPI_GATHER_INIT, <u>250</u> MPI_CATHERV_181	44
MPI_GATHERV, <u>181</u> MPI_CATHERV_INIT_251	45
MPI_GATHERV_INIT, <u>251</u> MPI_CET_525	46
MPI_GET, <u>525</u> MDI_CET_ACCUMULATE_521	47
MPI_GET_ACCUMULATE, <u>531</u>	
MPI_GET_ADDRESS, <u>129</u>	48

1	MPI_GET_COUNT, 35
2	MPI_GET_ELEMENTS, <u>141</u>
3	MPI_GET_ELEMENTS_X, <u>141</u>
4	MPI_GET_LIBRARY_VERSION, <u>408</u>
	MPI_GET_PROCESSOR_NAME, <u>410</u>
5	MPI_GET_VERSION, 407
6	MPI_GRAPH_CREATE, <u>358</u>
7	MPI_GRAPH_GET, <u>368</u>
8	$MPI_GRAPH_MAP, \frac{379}{379}$
9	MPI_GRAPH_NEIGHBORS, <u>371</u>
10	MPI_GRAPH_NEIGHBORS_COUNT, <u>371</u>
11	MPI_GRAPHDIMS_GET, <u>367</u>
	MPI_GREQUEST_COMPLETE, <u>585</u>
12	MPI_GREQUEST_START, <u>582</u> , 813, 954
13	MPI_GROUP_C2F, <u>781</u>
14	MPI_GROUP_COMPARE, <u>281</u>
15	MPI_GROUP_DIFFERENCE, <u>284</u>
16	$MPI_GROUP_EXCL, 285$
17	MPI_GROUP_F2C, <u>781</u>
18	MPI_GROUP_FREE, <u>288</u> , 451, 458
	MPI_GROUP_FROM_SESSION_PSET, <u>288</u>
19	MPI_GROUP_INCL, <u>284</u>
20	MPI_GROUP_INTERSECTION, <u>283</u>
21	MPI_GROUP_RANGE_EXCL, <u>287</u>
22	MPI_GROUP_RANGE_INCL, <u>286</u>
23	MPI_GROUP_RANK, <u>280</u>
24	MPI_GROUP_SIZE, <u>280</u>
25	MPI_GROUP_TRANSLATE_RANKS, 281
26	MPI_GROUP_UNION, <u>283</u>
	MPI_IALLGATHER, <u>236</u>
27	MPI_IALLGATHERV, 237
28	MPI_IALLREDUCE, 242
29	MPI_IALLTOALL, 238
30	MPI_IALLTOALLV, 239
31	MPI_IALLTOALLW, <u>240</u>
32	MPI_IBARRIER, <u>230</u> , 269
33	MPI_IBCAST, <u>230</u>
34	MPI_IBSEND, 57
	MPI_IEXSCAN, 246
35	MPI_IGATHER, 232
36	MPI_IGATHERV, 233
37	MPI_IMPROBE, 79
38	MPI_IMRECV, 83
39	MPI_INEIGHBOR_ALLGATHER, <u>391</u>
40	MPI_INEIGHBOR_ALLGATHERV, <u>392</u>
41	MPI_INEIGHBOR_ALLTOALL, <u>393</u>
42	MPI_INEIGHBOR_ALLTOALLV, <u>394</u>
	MPI_INEIGHBOR_ALLTOALLW, 395
43	MPI_INFO_C2F, <u>781</u>
44	$MPI_INFO_CREATE, \frac{436}{2}$
45	$MPI_INFO_CREATE_ENV, 441$
46	$MPI_INFO_DELETE, 437$
47	MPI_INFO_DUP, <u>441</u>
48	MPI_INFO_F2C, <u>781</u>
-	

MPI_INFO_FREE, <u>441</u>, 451, 458 MPI_INFO_GET, <u>437</u>, <u>724</u> MPI_INFO_GET_NKEYS, 440 MPI_INFO_GET_NTHKEY, 440 MPI_INFO_GET_STRING, 439 MPI_INFO_GET_VALUELEN, 438, 725 MPI_INFO_SET, 436 MPI_INIT, <u>444</u>, 672, 674, 682 MPI_INIT_THREAD, 447, 672, 674, 682 MPI_INITIALIZED, 453 MPI_INTERCOMM_CREATE, 323 MPI_INTERCOMM_CREATE_FROM_GROUPS, 324 MPI_INTERCOMM_MERGE, 325 MPI_IPROBE, 76 MPI_IRECV, 59 MPI_IREDUCE, 241 MPI_IREDUCE_SCATTER, 244 MPI_IREDUCE_SCATTER_BLOCK, 243 MPI_IRSEND, 59 MPI_IS_THREAD_MAIN, 449 MPI_ISCAN, 245 MPI_ISCATTER, 234 MPI_ISCATTERV, 235 MPI_ISEND, 57 MPI_ISENDRECV, 60 MPI_ISENDRECV_REPLACE, 61 MPI_ISSEND, 58 MPI_KEYVAL_CREATE, 721 MPI_KEYVAL_FREE, 722 MPI_LOOKUP_NAME, 495 MPI_MESSAGE_C2F, 781, 949 MPI_MESSAGE_F2C, 781, 949 MPI_MPROBE, 81 MPI_MRECV, 82 MPI_NEIGHBOR_ALLGATHER, 381, 942 MPI_NEIGHBOR_ALLGATHER_INIT, 396 MPI_NEIGHBOR_ALLGATHERV, 383, 942 MPI_NEIGHBOR_ALLGATHERV_INIT, 397 MPI_NEIGHBOR_ALLTOALL, 385, 942 MPI_NEIGHBOR_ALLTOALL_INIT, 398 MPI_NEIGHBOR_ALLTOALLV, 387, 942 MPI_NEIGHBOR_ALLTOALLV_INIT, 399 MPI_NEIGHBOR_ALLTOALLW, 389, 942 MPI_NEIGHBOR_ALLTOALLW_INIT, 401 MPI_NULL_COPY_FN, 722 MPI_NULL_DELETE_FN, 722 MPI_OP_C2F, 781 MPI_OP_COMMUTATIVE, 220 MPI_OP_CREATE, 213, 811 MPI_OP_F2C, 781 MPI_OP_FREE, <u>216</u>, 451, 458 MPI_OPEN_PORT, 489 MPI_PACK, <u>160</u>

MPI_PACK_EXTERNAL, 167 MPI_PACK_EXTERNAL_SIZE, 168 MPI_PACK_SIZE, 163 MPI_PARRIVED, 101 MPI_PCONTROL, 667 MPI_PREADY, <u>98</u> MPI_PREADY_LIST, 100 MPI_PREADY_RANGE, 99 MPI_PRECV_INIT, 97 MPI_PROBE, 77 MPI_PSEND_INIT, 96 MPI_PUBLISH_NAME, $\underline{493}$ MPI_PUT, <u>523</u> MPI_QUERY_THREAD, 449 MPI_RACCUMULATE, 538 MPI_RECV, 33, 270 MPI_RECV_INIT, 89 MPI_REDUCE, 204MPI_REDUCE_INIT, 260 MPI_REDUCE_LOCAL, 220 MPI_REDUCE_SCATTER, 223 MPI_REDUCE_SCATTER_BLOCK, 221 MPI_REDUCE_SCATTER_BLOCK_INIT, 262 MPI_REDUCE_SCATTER_INIT, 263 MPI_REGISTER_DATAREP, 644, 814 MPI_REQUEST_C2F, 781 MPI_REQUEST_F2C, 781 MPI_REQUEST_FREE, <u>65</u>, 451, 458 MPI_REQUEST_GET_STATUS, 75 MPI_RGET, <u>537</u> MPI_RGET_ACCUMULATE, 539 MPI_RPUT, <u>536</u> MPI_RSEND, 47 MPI_RSEND_INIT, 88 MPI_SCAN, <u>224</u> MPI_SCAN_INIT, 264 MPI_SCATTER, 189 MPI_SCATTER_INIT, 252 MPI_SCATTERV, 191 MPI_SCATTERV_INIT, 253 MPI_SEND, <u>28</u>, 270, 668 MPI_SEND_INIT, 86 MPI_SENDRECV, 38 MPI_SENDRECV_REPLACE, 39 MPI_SESSION_C2F, 781 MPI_SESSION_CALL_ERRHANDLER, 432 MPI_SESSION_CREATE_ERRHANDLER, 416, 422, 813 MPI_SESSION_F2C, 781 MPI_SESSION_FINALIZE, 457 MPI_SESSION_GET_ERRHANDLER, 416, 424 MPI_SESSION_GET_INFO, 461

MPI_SESSION_GET_NTH_PSET, <u>460</u>	1
MPI_SESSION_GET_NUM_PSETS, <u>459</u>	2
MPI_SESSION_GET_PSET_INFO, <u>462</u>	3
MPI_SESSION_INIT, <u>456</u>	4
MPI_SESSION_SET_ERRHANDLER, 416,	5
423	
MPI_SIZEOF, $\underline{726}$	6
MPI_SSEND, 46	7
MPI_SSEND_INIT, <u>87</u>	8
MPI_START, 89	9
MPI_STARTALL, 90	10
MPI_STATUS_C2F, 783	1
MPI_STATUS_C2F08, 784, 951	1:
MPI_STATUS_F082C, 783, 951	
$MPI_STATUS_F082F, 784$	13
MPI_STATUS_F2C, 782	1^{4}
MPI_STATUS_F2F08, <u>784</u>	15
MPI_STATUS_SET_CANCELLED, 589	16
MPI_STATUS_SET_ELEMENTS, 588	17
MPI_STATUS_SET_ELEMENTS_X, 589	18
MPI_T_CATEGORY_CHANGED, 718	19
MPI_T_CATEGORY_GET_CATEGORIES,	
717, 720	20
MPI_T_CATEGORY_GET_CVARS, 716, 720	2
MPI_T_CATEGORY_GET_EVENTS, 717	22
MPI_T_CATEGORY_GET_INDEX, 716, 720	23
MPI_T_CATEGORY_GET_INFO, 715, 720,	24
946	25
MPI_T_CATEGORY_GET_NUM, 714	26
MPI_T_CATEGORY_GET_NUM_EVENTS,	27
715	
MPI_T_CATEGORY_GET_PVARS, 717, 720	28
MPI_T_CVAR_GET_INDEX, <u>680</u> , 720	29
MPI_T_CVAR_GET_INFO, <u>679</u> , 720, 946	30
MPI_T_CVAR_GET_NUM, 678	3
MPI_T_CVAR_HANDLE_ALLOC, <u>682</u> , 720	33
MPI_T_CVAR_HANDLE_FREE, <u>683</u> , 720	33
MPI_T_CVAR_READ, <u>683</u> , 720	34
MPI_T_CVAR_WRITE, <u>683</u> , 720	
MPI_T_ENUM_GET_INFO, <u>677</u> , 720	35
MPI_T_ENUM_GET_ITEM, 677, 720	30
MPI_T_EVENT_CALLBACK_GET_INFO,	3'
708	38
MPI_T_EVENT_CALLBACK_SET_INFO,	39
708	40
MPI_T_EVENT_COPY, 712	41
MPI_T_EVENT_GET_INDEX, <u>704</u> , 720	
MPI_T_EVENT_GET_INFO, <u>703</u> , 720	43
MPI_T_EVENT_GET_NUM, <u>702</u>	4:
MPI_T_EVENT_GET_SOURCE, <u>713</u>	
	44
MPI T EVENT GET TIMESTAMP 712	44 45
MPI_T_EVENT_GET_TIMESTAMP, 712 MPI_T_EVENT_HANDLE_ALLOC, 705, 720	
MPI_T_EVENT_HANDLE_ALLOC, <u>705</u> , 720	4
	43 46

1	MPI_T_EVENT_HANDLE_SET_INFO, 706
2	MPI_T_EVENT_READ, 711
3	MPI_T_EVENT_REGISTER_CALLBACK,
4	$\frac{707}{1000}$
5	MPI_T_EVENT_SET_DROPPED_HANDLER,
6	710 MPI_T_FINALIZE, 675
7	MPI_T_INIT_THREAD, <u>674</u>
8	MPI_T_PVAR_GET_INDEX, <u>689</u> , 720
9	MPI_T_PVAR_GET_INFO, <u>688</u> , 720, 946
10	MPI_T_PVAR_GET_NUM, 687
11	MPI_T_PVAR_HANDLE_ALLOC, <u>691</u> , 720
12	MPI_T_PVAR_HANDLE_FREE, <u>692</u> , 720,
13	945
14	MPI_T_PVAR_READ, <u>693</u> , 720, 945
15	MPI_T_PVAR_READRESET, <u>694</u> , 720, 945 MPI_T_PVAR_RESET, <u>694</u> , 720, 945
16	MPI_1_PVAR_RESE1, <u>094</u> , 720, 945 MPI_T_PVAR_SESSION_CREATE, <u>690</u> , 720
17	MPI_T_PVAR_SESSION_FREE, 690, 720
18	MPI_T_PVAR_START, <u>692</u> , 720, 945
19	MPI_T_PVAR_STOP, <u>692</u> , 720, 945
20	MPI_T_PVAR_WRITE, <u>693</u> , 720, 945
20	MPI_T_SOURCE_GET_INFO, <u>699</u> , 720
21	MPI_T_SOURCE_GET_NUM, <u>698</u> , 720
	MPI_T_SOURCE_GET_TIMESTAMP, <u>700</u> ,
23	720 NDI TECE 64
24	MPI_TEST, <u>64</u> MPI_TEST_CANCELLED_85
25	MPI_TEST_CANCELLED, <u>85</u> MPI_TESTALL, <u>71</u> , 474, 583–585, 588
26	MPI_TESTANY, <u>69</u> , 474, 583, 584, 588
27	MPI_TESTSOME, <u>73</u> , 474, 583–585, 588
28	MPI_TOPO_TEST, <u>367</u>
29	MPI_TYPE_C2F, <u>780</u>
30	MPI_TYPE_COMMIT, <u>137</u>
31	MPI_TYPE_CONTIGUOUS, 111
32	MPI_TYPE_CREATE_DARRAY, <u>124</u>
33	MPI_TYPE_CREATE_F90_COMPLEX, <u>751</u> ,
34	753 MPI_TYPE_CREATE_F90_INTEGER, 751,
35	753
36	MPI_TYPE_CREATE_F90_REAL, <u>750</u> , 753
37	MPI_TYPE_CREATE_HINDEXED, 116
38	MPI_TYPE_CREATE_HINDEXED_BLOCK,
39	<u>118</u>
40	MPI_TYPE_CREATE_HVECTOR, <u>114</u>
41	MPI_TYPE_CREATE_INDEXED_BLOCK,
42	$\frac{118}{1000000000000000000000000000000000$
43	MPI_TYPE_CREATE_KEYVAL, 329, <u>340</u> , 343, 787, 812, 956
44	MPI_TYPE_CREATE_RESIZED, 135
45	MPI_TYPE_CREATE_STRUCT, <u>119</u> , 120
46	
	MPI_TYPE_CREATE_SUBARRAY, 121
47	MPI_TYPE_CREATE_SUBARRAY, <u>121</u> MPI_TYPE_DELETE_ATTR, <u>329</u> , <u>343</u>
47 48	

MPI_TYPE_DUP_FN, <u>340</u>, 946 MPI_TYPE_F2C, 780 MPI_TYPE_FREE, 138, 451, 458 MPI_TYPE_FREE_KEYVAL, 329, 341, 343 MPI_TYPE_GET_ATTR, 329, 342, 343, 740, 788 MPI_TYPE_GET_CONTENTS, 146 MPI_TYPE_GET_ENVELOPE, 144 MPI_TYPE_GET_EXTENT, 134 MPI_TYPE_GET_EXTENT_X, 134 MPI_TYPE_GET_NAME, 348 MPI_TYPE_GET_TRUE_EXTENT, 136 MPI_TYPE_GET_TRUE_EXTENT_X, 136 MPI_TYPE_INDEXED, 115 MPI_TYPE_MATCH_SIZE, 755 MPI_TYPE_NULL_COPY_FN, 340, 946 MPI_TYPE_NULL_DELETE_FN, 340, 946 MPI_TYPE_SET_ATTR, 329, 342, 343, 740, 788, 791 MPI_TYPE_SET_NAME, 348 MPI_TYPE_SIZE, 131, 668 MPI_TYPE_SIZE_X, 131 MPI_TYPE_VECTOR, 112 MPI_UNPACK, 161 MPI_UNPACK_EXTERNAL, 168 MPI_UNPUBLISH_NAME, 494 MPI_WAIT, 63, 270 MPI_WAITALL, 70, 271, 474, 583–585, 588 MPI_WAITANY, <u>68</u>, 474, 583, 584, 588 MPI_WAITSOME, 72, 474, 583-585, 588 MPI_WIN_ALLOCATE, 509 MPI_WIN_ALLOCATE_SHARED, 511 MPI_WIN_ATTACH, 516 MPI_WIN_C2F, 781 MPI_WIN_CALL_ERRHANDLER, 431 MPI_WIN_COMPLETE, 547 MPI_WIN_CREATE, 507 MPI_WIN_CREATE_DYNAMIC, 515 MPI_WIN_CREATE_ERRHANDLER, 416, 419, 813 MPI_WIN_CREATE_KEYVAL, 329, 336, 343, 787, 812, 956 MPI_WIN_DELETE_ATTR, 329, 339, 343 MPI_WIN_DETACH, 517 MPI_WIN_DUP_FN, <u>336</u>, 946 MPI_WIN_F2C, 781 MPI_WIN_FENCE, 544 MPI_WIN_FLUSH, 554 MPI_WIN_FLUSH_ALL, 555 MPI_WIN_FLUSH_LOCAL, 555 MPI_WIN_FLUSH_LOCAL_ALL, 556 MPI_WIN_FREE, 451, 458, 518 MPI_WIN_FREE_KEYVAL, 329, 338, 343

MPI_WIN_GET_ATTR, 329, 338, 343, 740, 788, 791 MPI_WIN_GET_ERRHANDLER, 416, 420, 957 MPI_WIN_GET_GROUP, 520 MPI_WIN_GET_INFO, 521MPI_WIN_GET_NAME, 349 MPI_WIN_LOCK, 551MPI_WIN_LOCK_ALL, 552 MPI_WIN_NULL_COPY_FN, 336, 946 MPI_WIN_NULL_DELETE_FN, <u>337</u>, 946 $MPI_WIN_POST, 548$ MPI_WIN_SET_ATTR, 329, 338, 343, 740, 788, 791 MPI_WIN_SET_ERRHANDLER, 416, 420 MPI_WIN_SET_INFO, 521 MPI_WIN_SET_NAME, <u>349</u> MPI_WIN_SHARED_QUERY, 513 MPI_WIN_START, 547MPI_WIN_SYNC, 556 MPI_WIN_TEST, 550 MPI_WIN_UNLOCK, 552MPI_WIN_UNLOCK_ALL, 553 MPI_WIN_WAIT, 549MPI_WTICK, $\underline{434}$ MPI_WTIME, <u>433</u>, 668 mpiexec, 448, 469, 470 mpirun, 470

PMPI_ISEND, 740, 743

1

2

3

4

5

6

7

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

 24

25 26